University of West Attica

Department of Informatics and Computer Engineering

DIPLOMA THESIS

Parallel Classification using Support Vector Machines on Modern GPUs

Evangelos Katsandris

Supervisor: Mamalis Vasileios

March 2024

Πανεπιστήμιο Δυτικής Αττικής

Τμήμα Μηχανικών Πληροφορικής και Υπολογιστών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παράλληλη Κατηγοριοποίηση Δεδομένων με χρήση Μηχανών Διανυσμάτων Υποστήριξης (SVMs) σε Περιβάλλον Σύγχρονων Καρτών Γραφικών (GPUs)

Ευάγγελος Κατσανδρής

Επιβλέπων: Μάμαλης Βασίλειος

Μάρτιος 2024

Επιτροπή Εξέτασης

Καρκαζής Παναγιώτης	Μάμαλης Βασίλειος	Μπόγρης Αντώνιος
Panagiotis Karkazis Digitally signed by Panagiotis Karkazis Date: 2024.04.02 09:43:58 +03'00'	Vasileios Mamalis Digitally signed by Vasileios Mamalis Date: 2024.03.31 16:44:24 +03'00'	Antonios Bogris Digitally signed by Antonios Bogris Date: 2024.04.01 14:27:55 +03'00'
Αν. Καθηγητής	Καθηγητής	Καθηγητής

Ημερομηνία Εξέτασης: 22/03/2024

ΔΗΛΩΣΗ ΣΥΓΓΡΑΦΕΑ ΔΙΠΛΩΜΑΤΙΚΗΣ ΕΡΓΑΣΙΑΣ

Ο κάτωθι υπογεγραμμένος Κατσανδρής Ευάγγελος του Εμμανουήλ, με αριθμό μητρώου 711171014 φοιτητής του Τμήματος Μηχανικών Πληροφορικής και Υπολογιστών της Σχολής Μηχανικών του Πανεπιστημίου Δυτικής Αττικής, δηλώνω ότι:

«Βεβαιώνω ότι είμαι συγγραφέας αυτής της Διπλωματικής εργασίας και κάθε βοήθεια την οποία είχα για την προετοιμασία της, είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, οι όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε ακριβώς είτε παραφρασμένες, αναφέρονται στο σύνολό τους, με πλήρη αναφορά στους συγγραφείς, τον εκδοτικό οίκο ή το περιοδικό, συμπεριλαμβανομένων και των πηγών που ενδεχομένως χρησιμοποιήθηκαν από το διαδίκτυο. Επίσης, βεβαιώνω ότι αυτή η εργασία έχει συγγραφεί από μένα αποκλειστικά και αποτελεί προϊόν πνευματικής ιδιοκτησίας τόσο δικής μου, όσο και του Ιδρύματος. Παράβαση της ανωτέρω ακαδημαϊκής μου ευθύνης αποτελεί ουσιώδη λόγο για την ανάκληση του πτυχίου μου».

Ο Δηλών

Κατσανδρής Ευάγγελος

Acknowledgements

I would like to express my deepest gratitude to my parents for their continued love, support, and encouragement throughout my academic journey. Their sacrifices and belief in me have been the cornerstone of my achievements.

I am also grateful to my brother for his work ethic, which allowed my parents to more easily support me through my studies, and for being a source of inspiration in pursuing my goals.

Special thanks are due to, Prof. Vasilios Mamalis, for his invaluable guidance, and unwavering patience. Their mentorship, constructive feedback, and encouragement have been instrumental in shaping this thesis.

Last but not least, I am grateful to my friends and extended family for their patience, understanding, and encouragement during this challenging yet rewarding journey.

Περίληψη

Αντικείμενο της διπλωματικής εργασίας θα είναι ο σχεδιασμός, ανάπτυξη και αξιολόγηση σε περιβάλλον προγραμματισμού CUDA ενός αποδοτικού παράλληλων αλγορίθμου SVM για κατηγοριοποίηση δεδομένων. Η ανάπτυξη του ανωτέρω αλγορίθμου θα γίνει σε γλώσσα C/C++, ενώ η αξιολόγησή του θα πραγματοποιηθεί σε πραγματικό περιβάλλον σύγχρονων καρτών γραφικών (NVIDIA 1060 και NVIDIA RTX TITAN) και θα περιλαμβάνει σύγκριση (σε επίπεδο χρόνων απόκρισης, επιτάχυνσης-speedup και ακρίβειας της κατηγοριοποίησης/accuracy) με μια αντίστοιχή υλοποιήση σε σειριακό περιβάλλον εκτέλεσης.

Επιστημονική Περιοχή: Παράλληλα Υπολογιστικά Συστήματα

Λέξεις κλειδιά: SVM, CUDA, GPGPU, Παράλληλος Προγραμματισμός, Κατηγοριοποίηση Δεδομένων

Abstract

The objective of this thesis will be the design, development and evaluation of an efficient parallel SVM classifier in a CUDA programming environment. The development of the above classifier will be done using C/C++, while the evaluation will be performed in a real, modern environment using modern GPUs (NVIDIA 1060 και NVIDIA RTX TITAN) and will include comparisons (latency, speedup and accuracy of classification) with an equivalent sequential implementation.

Scientific Field: Parallel Computing Systems

Keywords: SVM, CUDA, GPGPU, Parallel Programming, Data Classification

Table of contents

ΔΗΛΩΣΗ ΣΥΓΓΡΑΦΕΑ ΔΙΠΛΩΜΑΤΙΚΗΣ ΕΡΓΑΣΙΑΣ 4

Acknowledgements 5

Περίληψη 6

Abstract 7

Glossary 14

1 Introduction 15

2 Classification 16

2.1 Learning Strategies . . . 16

2.1.1 Batch Learning . . . 16

2.1.2 Online learning . . . 17

2.2 Binary and Multiclass . . . 17

2.2.1 One vs All . . . 17

2.2.2 One vs One . . . 18

2.3 Linear Classifiers . . . 18

2.4 Classifiers . . . 18

2.4.1 Perceptron . . . 19

2.4.2 Neural Networks . . . 20

2.4.3 K-Nearest Neighbor . . . 20

2.4.4 Naive Bayes . . . 21

2.4.5 SVM . . . 23

2.5 Sequential Minimization Optimization . . . 34

3 CUDA 36

3.1 Architecture of Nvidia GPUs . . . 36

3.2 Programming Model . . . 37
3.2.1 Execution and Threading Model . . . 37
3.2.2 Memory . . . 40
3.2.3 Compilation . . . 41

4 Parallelization of SVM **43**
4.1 Parallelization Techniques . . . 43
4.2 Parallel SVM Algorithms: . . . 43
4.3 P-SMO . . . 43
4.3.1 Results . . . 45
4.4 Parallel-Parallel SMO . . . 46
4.4.1 Results . . . 47
4.5 GPUSVM . . . 47
4.5.1 Results . . . 47
4.6 PCV . . . 48
4.6.1 Results . . . 48
4.7 SVM-SMO-SDG . . . 48
4.7.1 Results . . . 49
4.8 C-SVM . . . 49
4.8.1 Results . . . 50
4.9 ECM . . . 50
4.9.1 Results . . . 50

5 Implementation **51**
5.1 CLI Interface . . . 51
5.2 Serial SMO . . . 52
5.3 Parallel GPUSVM . . . 52
5.4 Vector Library . . . 53

6 Experimental Results **54**
6.1 Hardware . . . 54
6.2 Datasets . . . 55
6.2.1 Linear . . . 55
6.2.2 Iris . . . 55
6.3 Baseline SMO vs GPUSVM . . . 55
6.4 Varying Number of Threads . . . 55

6.5 Small Datasets 56

7 Conclusions and Future work **58**

References **59**

Appendix **63**

7.1 Full Project 73

List of Figures

2.4.1 Maximum-margin hyperplane and margin for an SVM trained on two classes. Larhmam / CC BY-SA 4.0 DEED 23

3.2.1 Grid of Thread Blocks 38

3.2.2 Memory Hierarchy 41

4.8.1 Architecture of a binary Cascade SVM [43] 50

6.3.1 Training time of serial SMO and parallel GPUSVM 56

6.4.1 Training time for differing thread count 57

6.5.1 Training time for small datasets by algorithm 57

List of Tables

2.1 Kernels used with SVM 31

6.1 WSL System 54
6.2 Headless System 54

7.1 Raw Data collected for “Training time of serial SMO and parallel GPUSVM” 64
7.1 Raw Data collected for “Training time of serial SMO and parallel GPUSVM” 65
7.1 Raw Data collected for “Training time of serial SMO and parallel GPUSVM” 66
7.1 Raw Data collected for “Training time of serial SMO and parallel GPUSVM” 67
7.1 Raw Data collected for “Training time of serial SMO and parallel GPUSVM” 68
7.2 Raw Data collected for “Training time for differing thread count” 68
7.2 Raw Data collected for “Training time for differing thread count” 69
7.2 Raw Data collected for “Training time for differing thread count” 70
7.3 Raw Data collected for “Training time for small datasets by algorithm” 70
7.3 Raw Data collected for “Training time for small datasets by algorithm” 71
7.3 Raw Data collected for “Training time for small datasets by algorithm” 72
7.3 Raw Data collected for “Training time for small datasets by algorithm” 73

List of Listings

3.1 CUDA Kernel Invocation 38
3.2 CUDA Addition Kernel 39

4.1 P-SMO Pseudocode 46
4.2 GPUSVM Pseudocode 47

5.1 Usage of the cli interface 52
5.2 Calculation of optimal number of blocks 53

7.1 Generic Vector Implementation 63
7.2 Generic Vector Implementation cont. 74
7.3 Generic Vector Implementation cont. 75
7.4 Generic Vector Implementation cont. 76
7.5 Generic Matrix Implementation 77
7.6 Generic Matrix Implementation cont. 78
7.7 Generic Matrix Implementationt cont. 79
7.8 SMO Implementation Outer Loop 80
7.9 SMO Implementation Inner Loop 81
7.10 SMO Implementation Step 82
7.11 GPUSVM Grid Reduction 83
7.12 GPUSVM Grid Reduction cont. 84
7.13 GPUSVM training device-side code 85
7.14 GPUSVM training device-side code cont. 86
7.15 GPUSVM training device-side code cont. 87
7.16 Linearly separable dataset generation script 87

Glossary

Term	Definition
CUDA	Compute Unified Device Architecture: A GPGPU platform and library.
Classifier	An algorithm that assigns class labels to data.
GPGPU	General Purpose GPU computing: The use of GPUs for accelarion of tasks other than image processing.
GPU	Graphics Processing Unit: A hardware accelarator designed for use in image processing.
OVA	One Versus All: A technique for using binary classifiers for multiclass classification, by training binary classifiers to separate samples into a specific class or one of the rest.
OVO	One Versus One: A technique for using binary classifiers for multiclass classification, by training binary classifiers to separate samples into one of two classes and picking the class most picked.
OVR	One Versus Rest: see OVA.
PSMO	Parallel SMO: A variant of SMO meant for use in parallel computing environments.
QP	Quadratic Programming: A family of techniques for solving optimization problems involving quadratic functions.
SMO	Sequential Minimization Optimization: An SVM algorithm based on coordinate descent instead of reliying upon standard QP techniques.
SVM	Support Vector Machine: a data classification algorithm
support vector	Samples on the margin of the separating hyperplane of an SVM.

1 Introduction

SVM is a very powerful algorithm used in the classification of both linearly separable and non-linearly separable data, however the computing power required to train an SVM model is high. Many advancements have been made since it's inception [6] such as the use of the Sequential Minimal Optimization (SMO) algorithm [7], instead of using quadratic programming techniques, and parallelization using PSVM [8]. This paper will make an attempt at leveraging the PSVM and other improvements in a heterogeneous CUDA computing environment. This will be achieved by taking advantage of the inherent parallelism present in PSVM due to the division of the dataset into smaller independent subsets which maps very well to the CUDA programming model. The paper will go over the basics of data classification, existing improvements on SVMs as well as the CUDA programming model. It will examine previous work done to improve the efficiency of SVM, including algorithm as well as implementation specific improvements pertaining to CUDA. An initial naive non-parallel implementation of SVM will be provided as a benchmark and all steps taken to optimize it will be evaluated for their contribution to the overall speedup. The main goal of this paper will be the implementation of an efficient parallelized SVM algorithm that will take full advantage of the heterogeneous computing environment it's run on to achieve fast training and prediction times.

2 Classification

Classification is the process of assigning a class label to each sample of a dataset, an algorithm that classifies data is called a Classifier. Classification has a wide range of applications ranging from medical diagnosis [9] to natural language processing [10] and building recognition from aerial photography [11] to financial fraud detection [12]. As such lots of research has gone into improving Classifiers. This section will cover the training of classifiers, various categories of classifiers and finally various types of classifiers used today.

2.1 Learning Strategies

Classification is in it's simplest form can be split into two distinct phases, the training phase: wherein the classifier learns how to classify samples, and the prediction phase: wherein the classifier actually attempts to classify distinct samples. But with more involved techniques and depending on when the training phase happens there exist multiple training strategies.

2.1.1 Batch Learning

In batch training, the training and prediction phases **are** distinct and the training happens all at once—before the prediction stage.

Supervised learning

The datasets used in training are usually annotated with the class label of each data point. In supervised learning [13], the dataset used in training will be annotated with the class label of each sample. In this way the classifier can learn to extract sets of features that are representative of each class by summarizing the common features of samples with the same class label.

Unsupervised learning

Techniques that can operate on unlabeled datasets exist and fall under the name of unsupervised learning [14]. In unsupervised learning, the dataset used in training is not annotated. Unsupervised learning techniques are used when a training dataset does not exist or is prohibitively expensive, time consuming or inconvenient to annotate correctly [15].

2.1.2 Online learning

With online learning, as opposed to Batch Learning, the classifier is designed to learn on $\ell + 1$ samples, where ℓ is the number of samples already having been used for training. In other words the classifier is given an unlabeled sample, it makes a prediction, and then the classifier is given the correct label which it uses to update it's model [16, page 20]. Online learning can be used in conjunction with either unsupervised and supervised learning. Online learning is useful when the training data is prohibitively large [17], is generated in real time [18] or in general anytime when training has to be interleaved with prediction.

2.2 Binary and Multiclass

Depending on the number of classes in a dataset there exist two kinds of classification. Binary classification is when the number of classes is exactly two and multiclass classification is for more than two classes. Not all kinds of classifiers do inherently support multiclass classification, but one can implement multiclass classifiers by using multiple binary classifiers. There are two main methods of extending binary classifiers to be used in multiclass classification.

2.2.1 One vs All

One vs All[1] (OVA) [19] multiclass classifiers have a classifier for each class. For n classes, n binary classifiers are required, one for each class. Each binary classifier distinguishes whether a sample belongs to one specific class or if it belongs to any of the rest classes. To assign the final class label to the sample the class label with the highest prediction confidence is used.

[1] One vs All: Also known as One vs Rest (OVR), One Against All (OAA) or One Against Rest (OAR)

2.2.2 One vs One

One vs One[2] (OVO) [19] multiclass classifiers have a classifier for every 2-combination[3] of classes, for n classes $\binom{n}{2}$ binary classifiers are required.

$$\binom{n}{2} = \frac{n(n-2+1)}{2(2-1)1} = \frac{n(n-1)}{2} \tag{2.1}$$

Each binary classifier labels whether a sample belongs to one of it's two classes classes. The final class label is assigned to a sample using the class label predicted by the majority of the binary classifiers.

2.3 Linear Classifiers

A linear classifier is a classifier that can only correctly predict the class of all it's inputs if the dataset is linearly separable. For a dataset to be linearly separable there must exist at least one hyperplane[4] that can create two half-spaces where each one only contains one class of samples.

more abstractly a set X with distinct subsets X_0 and X_1 is said to be linearly separable if there exist $n+1, w_1, w_2, ..., w_n, k$ where $w_i, k \in \mathbb{R}$ satisfying

$$\sum_{i=1}^{n} w_i x_i > k \tag{2.2}$$

$$\sum_{i=1}^{n} w_i x_i < k \tag{2.3}$$

where x_i is the i-th element of X_0 and X_1 respectively.

Non linear classifiers do not have this limitation.

2.4 Classifiers

This subsection will go over a number of classifiers, as well as their way of operation. It will also go over the basics of SVM, but more details will be given in Parallelization of SVM

[2] One vs one: Also known as One Against One (OAO)
[3] 2-combination: a distinct selection of 2 elements from a set
[4] a subset with dimension $n-1$ where n is the dimension of the dataset

2.4.1 Perceptron

The perceptron was first described as the McCulloch and Pitts neuron by Mcculloch and Pitts [20] in 1943. Based upon that work Rosenbaltt made one of the earliest attempts to replicate a biological neuron using digital circuits [21, circa 1957]. The effectiveness of the perceptron was criticized by Minsky and Papert [22] leading to the work being largely ignored and further research to stop until much later.

Today we would describe the perceptron as a binary linear classifier for use in supervised learning. As far as application of perceptron models go normal single layer perceptrons don't see much practical usage due to their simplicity and shortcommings involving non-linearly separable datasets, but they serve as the foundations of other models such as Multi-Layer Perceptrons or Neural Networks. They also see use in educational contexts exactly because of their simplicity to serve as an introduction to machine learning models.

In simple terms it's a simplified model of brain neurons. It works by evaluating a function $f(x)$:

$$f(x) = \begin{cases} 1, & \sum_{i=0}^{n} w_i \cdot x_i + b > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.4}$$

where w is a vector of n real valued weights $w_0, w_1, ...x_n$

x is a vector of n real valued samples $x_0, x_1, ...x_n$

n is the number of inputs to the perceptron,

and b is a bias term used to shift the activation boundary away from the origin.

2.4.1.1 Training Perceptrons

Training a perceptron involves a number of steps and the use of a label vector y of m elements $y_0, y_1...y_m$ corresponding to each sample vector x. First the values for the w and b are chosen at random. Afterwards, for an input vector x, each sample is multiplied with it's coresponding weight in w, and the results plus the b are summed up. Then the activation function $f(x)$ is used on the result and compaired to the coresponding label in y in order to calculate a prediction error using a loss function [23, pages 349-350]. Using the prediction error, w and b are adjusted. This is repeated for all m input vectors until the perceptron has reached a satisfactory prediction accuracy.

2.4.2 Neural Networks

Neural Networks are a wide category of models but in general can be simply thought of as a network of perceptrons [24]. Many variants exist such as Deep Neural Networks [25] for more complex classification problems and Convolutional Neural Networks [26] for classification of images. Being such a wide category, Neural Networks have wide applications in all classification tasks mentioned in Classification.

They are comprised by three types of layers:

1. The input layer, with n neurons, one for each input sample.
2. The hidden layers, with an arbitrary number of neurons for each layer.
3. The output layer, that can include:
    1. only one neuron for simple binary classification
    2. multiple ones for confidence based multiclass classification
    3. many more for use in other non-classification applications, as for example image generation where the output layer might include as many or more neurons as the input layer.

2.4.2.1 Training Neural Networks

The details of training a neural network depend on it's exact variant but the general case is as follows. First all the weights and bias term are chosen at random. Then the model is an input, with each sample going throught the network's neurons's activation functions. And finally the results are consolidated in the output layer. Now using backpropagation [27, pages 9-10] and a loss function [23, pages 349-350] the weights and biases of each neuron are recalculated. This is repeated for many inputs until a satisfactory model has been trained.

2.4.3 K-Nearest Neighbor

K-Nearest Neighbor also known as k-NN or KNN is an unsupervised learning technique for use in classification that works by finding the k nearest neighbors of an input vector in the feature space and classifies inputs (also called the queries) according the the most common label of the found neighbors. A "1"-Nearest Neightbor algorithm is descripbed by Cover and Hart [28]. Of note is the fact that training a k-NN model equals storing the training samples, a fact that restricts a naive implementation of the model based on the available memory and the size of the training dataset. Non-naive implementations include

k-D Trees [29] and Ball Trees [30] that attempt to deal with the inefficiencies of the algorithm. It is loosly related and not to be confused with the Kmeans clustering algorithm [31].

2.4.3.1 Training and Prediction with k-NNs

As mentioned before, training such a model is very simple as it only involves storing the training inputs and their coresponding labels. The important part is correctly choosing an k for the number of neighbors and the distance metric —both depend on the dataset used. Selecting the value for k requires the use of heuristic hyperparameter optimization techniques. Apart from hyperparameter optimization, it can also benefit from dimensionality reduction of the feature space. The distance metric can be any Minkowski distance or in the case of text classfication a Levenshtein distance [32] can be used, again it highly depends on the shape of the data. Afterwards for a chosen N, predicting the class of an input is as simple as finding the N closest training inputs

2.4.4 Naive Bayes

Naive Bayes is a classifier based on the Bayes Theorem 2.5 and the "naive" assumption that the features of the classes are independent of one another.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.5}$$

From 2.5 and according to Zhang [33]

we can say that the probability that a sample x with features $< x_1, x_2, ..., x_n >$ belongs in class c is:

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)} \tag{2.6}$$

And x is classified in class C iif:

$$f_b(x) = \frac{P(C = +|x)}{P(C = -|x)} \geq 1, \tag{2.7}$$

Again given the naive assumption that features are independent:

$$f_nb(x) = \frac{P(C = +|x)}{P(C = -|x)} \prod_{i=1}^{n} \frac{P(x_i|C = +)}{P(x_i|C = -)} \tag{2.8}$$

As such the function $f_nb(x)$ is the Naive Bayes Classifier.

2.4.4.1 Types of NB Classifiers

Depending on the feature values of the dataset, specific types of NB Classifiers exist.

- Gaussian Naive Bayes: is used when the feature values follow a guassian (ie. normal) distribution. For example the heights of people.
- Multinomial Naive Bayes: is used when the features values are distinct counts. For example word counts.
- Bernoulli Naive Bayes: is used when the feature values can only take on one of two values and have no ordering. For example when they are booleans.
- Categorical Naive Bayes: is used when the feature values are categorical, in other words they have no ordering. For example the color of a car.

2.4.4.2 Training and Prediction with Naive Bayes

A Naive Bayes model is trained thought calculating the probabilities needed. First, the a priori probabilities $P(c)$ of each class are calculated using their frequency of appearance in the training set:

$$P(c_i) = \frac{n_i}{N} \tag{2.9}$$

n_i is the number of samples of class c_i and N is the total number of samples.

Then, depending on the exact type of Naive Bayes used, the probability of each feature given a class is calculated [33].

A kernel function may be used to estimate the probabilities in order to improve it's performance much like it's used for dimensionallity reduction in

2.4.5 SVM

Support Vector Machines is a linear binary classifier that works by attempting to find the maximally separating hyperplane between two classes. A hyperplane is an $n-1$ dimensional vector where n is the dimensionality of the feature space.

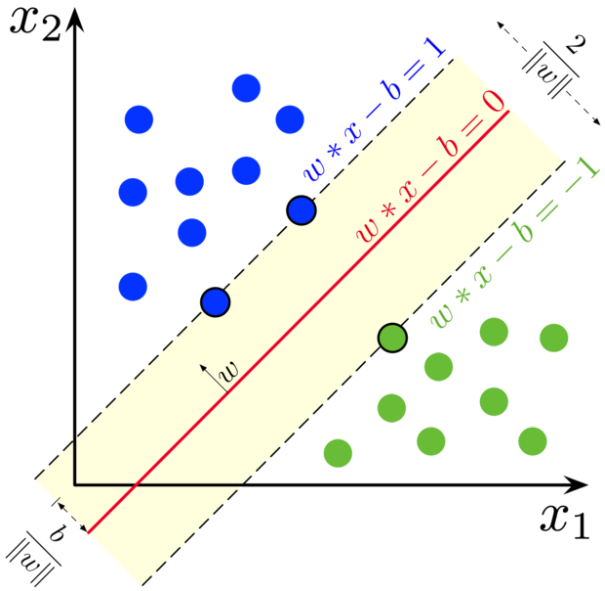

Figure 2.4.1: Maximum-margin hyperplane and margin for an SVM trained on two classes. Larhmam / CC BY-SA 4.0 DEED

2.4.5.1 The Primal Problem

More formally an SVM classifier tries to solve a constrained optimization problem. This section will now prove that SVM's problem is a constraint optimization problem. Let w to be the separating hyperplane, b a bias term and x a vector of n feature samples.

$$\min_{w,b} \frac{||W||^2}{2} \tag{2.10}$$

Subject to:

$$y_i(w^T + b) >= 1, \text{ for } i = 1, \ldots, n \tag{2.11a}$$

The problem can also be expressed as a general convex optimization problem of the following form:

$$\min_{x} f(x) \tag{2.12}$$

Subject to:

$$g_i(x) \leq 0, \text{ for } i = 1, \dots, n \tag{2.13a}$$

Substituting with:

$$f(w,b) = \frac{||W||^2}{2} \tag{2.14a}$$

$$g_i(w,b) = 1 - y_i(w^T x_i + b) \tag{2.14b}$$

As such the problem can finally be written as:

$$\min_{w,b} f(w,b), \text{where } f(w,b) = \frac{||W||^2}{2} \tag{2.15}$$

Subject to:

$$g_i(w,b) \leq 0 \tag{2.16a}$$

$$g_i(w,b) = 1 - y_i(w^T x_i + b) \tag{2.16b}$$

$$\tag{2.16c}$$

But for this to actually be a convex optimization problem, it will need to be shown that f and all g_i are convex functions.

For f it is sufficient to show that it's Hessian matrix is positive semidefinite (PSD). For a matrix to be PSD it is sufficient to show that $\forall z \; z^T H z >= 0$.

Proof. The Hessian matrix is defined as such:

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{2.17}$$

As such for f, it's Hessian is:

$$H = \begin{bmatrix} \frac{\partial 1/2||W||^2}{\partial w_1^2} & \cdots & \frac{\partial 1/2||W||^2}{\partial w_1 \partial w_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial 1/2||W||^2}{\partial w_d \partial w_1} & \cdots & \frac{\partial 1/2||W||^2}{\partial w_d \partial w_d} \end{bmatrix} \tag{2.18}$$

Knowing that $1/2||W||^2 = 1/2 \sum_{i=1}^{d} w_i^2$, the first partial derivative can be computed to get:

$$H = \begin{bmatrix} \frac{\partial w_1}{\partial w_1} & \cdots & \frac{\partial w_1}{\partial w_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial w_d}{\partial w_1} & \cdots & \frac{\partial w_d}{\partial w_d} \end{bmatrix} \tag{2.19}$$

Computing the second partial derivative gets us:

$$H = \begin{bmatrix} 1 & ... & 0 \\ \vdots & \ddots & \vdots \\ 0 & ... & 1 \end{bmatrix} = I \tag{2.20}$$

Which is obviously the identity matrix. As such:

$$\begin{aligned} z^T H z \geq & 0 \Rightarrow \\ z^T I z \geq & 0 \Rightarrow \\ z^T z \geq & 0 \Rightarrow \\ \sum_{i=1}^{d} z_i^2 \geq & 0 \end{aligned}$$

Which clearly holds true. □

Similarly it will shown that all g_i are convex, because their Hessians are PSD.

Proof.

$$H = \begin{bmatrix} \frac{\partial^2(1-y_i(w^T x_i+b))}{\partial w_1 \partial w_d} & \cdots & \frac{\partial^2(1-y_i(w^T x_i+b))}{\partial w_d \partial b} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2(1-y_i(w^T x_i+b))}{\partial b \partial w_1} & \cdots & \frac{\partial^2(1-y_i(w^T x_i+b))}{\partial b \partial b} \end{bmatrix} \Rightarrow \tag{2.21}$$

$$H = \begin{bmatrix} \frac{\partial -y_i x_{i,1}}{\partial w1} & \cdots & \frac{\partial -y_i x_{i,1}}{\partial b} \\ \vdots & \ddots & \vdots \\ \frac{\partial -y_i}{\partial w_1} & \cdots & \frac{\partial -y_i}{\partial b} \end{bmatrix} = I \Rightarrow \tag{2.22}$$

$$H = \begin{bmatrix} 0 & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & 0 \end{bmatrix} \tag{2.23}$$

For which $\forall z$, $z^T H z >= 0$ clearly holds true. □

2.4.5.2 The Dual Problem

The dual form of this problem will be used to solve the primal form of this optimization problem. Introducing the generalized Lagrangian, defined as:

$$L(x, z) = f(x) + \sum_{i=1}^{n} a_i g_i(x) \tag{2.24}$$

Where a_i are what are known as Lagrange multipliers.

Denote:

$$\begin{aligned} \theta_P(x) &= \max_{a:a_i \geq 0} L(x, a) \\ &= \max_{a:a_i \geq 0} f(x) + \sum_{i=1}^{n} a_i g_i(x) \end{aligned} \tag{2.25}$$

Assume that an x, violates one primal $g_j(x) > 0$ constraint. Since the maximum a is picked, let $a_j \to \infty$, to get $\theta_P(x) = \infty$.

Assume that an x, satisfies **all** primal $g_j(x) \leq 0$ constraint. Let all $a_j = 0$, and get $\theta_P(x) = f(x)$.

From this it is shown that if we assume that x satisfies all primal constraints that the following has the same optimal solution as the original primal problem:

$$\min_x \theta_P(x) = \min_x \max_{a:a_i \geq 0} L(x, a) \tag{2.26}$$

Let $p^* = \min_x \theta_P(x)$ denote the **primal value**.

A dual problem is defined as a function:

$$\theta_D(a) = \min_x L(x, a) \tag{2.27}$$

and a **dual value**:

$$\begin{aligned} d^* &= \max_{a:a_i \geq 0} \theta_D(a) \\ &= \max_{a:a_i \geq 0} \min_x L(x, a) \end{aligned} \tag{2.28}$$

The maximum of the minimum of something is obviously less than or equal to the minimum of the maximum of something, and as such we can see that $d^* \leq p^*$

Theorem 1. *If there exists an x^* that solves the primal problem and an (μ^*, λ^*) that solves the dual problem, such that they both satisfy the Karush-Kuhn-Tucker (KKT) conditions, then the problem is said to have strong duality. If the problem pair has strong duality, then for any solution x^* to the primal problem and any solution (μ^*, λ^*) to the dual problem, the pair $x^*, (\mu^*, \lambda^*)$ must satisfy the KKT conditions [34].*

The KKT conditions are as follows:

$$\begin{aligned} \frac{\partial L(x^*, a^*)}{\partial x_i} &= 0 \ \forall i \in 1, \dots, n \\ a_i^* g_i(x^*) &= 0 \ \forall i \in 1, \dots, n \\ g_i(x^*) &\leq 0 \ \forall i \in 1, \dots, n \\ a_i^* &\geq 0 \ \forall i \in 1, \dots, n \end{aligned} \tag{2.29}$$

If f and all g_i are convex and the g_i constraints are strictly feasible[5] it is trivial to prove that $d^* = p^*$.

[5] strictly feasible: $\exists x, g_i(x) < 0, \forall i$

We will now show that the g_i constraints are strictly feasible:

Proof. If we assume we have a linearly separable dataset then a separating hyperplane $w^T x + b$ should correctly classify all samples, in other words $y_i(w^T x + i + b) > 0 \forall i$.

As such we could scale w and b by an arbitrary number in order for $g_i(w, b) < 0$ to hold true, where $g_i(w, b) = 1 - y_i(w^T x_i + b)$. □

We will now attempt to solve the dual form of the problem:

$$\max_{a:a_i \geq 0} \min_{w,b} f(w, b) + \sum_{i=1}^{n} a_i g_i(w, b) \tag{2.30}$$

$$\max_{a:a_i \geq 0} \min_{w,b} 1/2||W||^2 - \sum_{i=1}^{n} a_i(y_i(w^T x_i + b) - 1) \tag{2.31}$$

From the first KKT condition in Equation 2.29 we know that all the partial derivatives of the Generalized Lagrangian will equal 0. Taking the partial derivative in respect to w_j:

$$\frac{\partial 1/2||W||^2 - \sum_{i=1}^{n} a_i(y_i(w^T x_i + b) - 1)}{\partial w_j} = 0 \Rightarrow \tag{2.32}$$

$$w_j - \sum_{i=1}^{n} a_i y_i x_{i,j} = 0 \Rightarrow \tag{2.33}$$

$$w = \sum_{i=1}^{n} a_i y_i x_{i,j} \tag{2.34}$$

Substituting Equation 2.34 into Equation 2.31:

$$\max_{a:a_i \geq 0} \min_{w,b} 1/2||W||^2 - \sum_{i=1}^{n} a_i(y_i((\sum_{j=1}^{n} a_j y_j x_j)^T x_i + b) - 1) \tag{2.35}$$

Expanding it:

$$\max_{a:a_i \geq 0} \min_{w,b} 1/2||\sum_{i=1}^{n} a_i y_i x_i||^2 - \sum_{i=1}^{n} a_i(y_i((\sum_{j=1}^{n} a_j y_j x_j)^T x_i + b) - 1) \tag{2.36}$$

$$\max_{a:a_i \geq 0} \min_{w,b} 1/2 \sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j y_i y_j x_i^T x_j - \sum_{i=1}^{n} a_i(y_i((\sum_{j=1}^{n} a_j y_j x_j)^T x_i + b) - 1) \tag{2.37}$$

$$\max_{a:a_i \geq 0} \min_{w,b} \sum_{i=1}^{n} a_i - 1/2 \sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j y_i y_j x_i^T x_j - b\sum_{i=1}^{n} a_i y_i \tag{2.38}$$

Again from the first KKT condition in Equation 2.29 we know that all the partial derivatives of the Generalized Lagrangian will equal 0. Now taking the partial derivative in respect to b:

$$\frac{\partial 1/2||W||^2 - \sum_{i=1}^{n} a_i(y_i(w^T x_i + b) - 1)}{\partial b} = 0 \Rightarrow \tag{2.39}$$

$$-\sum_{i=1}^{n} a_i y_i = 0 \tag{2.40}$$

Substituting Equation 2.40 into Equation 2.38:

$$\max_{a:a_i \geq 0} \min_{w,b} \sum_{i=1}^{n} a_i - 1/2 \sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j y_i y_j x_i^T x_j \tag{2.41}$$

Note that Equation 2.41 no longer includes neither w, nor b, so it can be safely expressed without the minimum. As such we finally get the dual form of the SVM problem that we can solve:

$$\max_{a:a_i \geq 0} \sum_{i=1}^{n} a_i - 1/2 \sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j y_i y_j x_i^T x_j \tag{2.42}$$

Subject to:

$$\sum_{i=1}^{n} a_i y_i = 0 \tag{2.43}$$

Recovering optimal bias parameter

The optimal value for b must be one that pushes the separating hyperplane to sit between the furthest support vector in w's direction and the closest support vector of the other class. In other words their functional margins $y_i(w^T x_i + b)$ must be equal:

$$\min_{i:y_i=1} w^T x_i + b = -(\max_{i:y_i=-1} w^T x_i + b) \Rightarrow \tag{2.44}$$

$$b = -\frac{1}{2}(\min_{i:y_i=1} w^T x_i + \max_{i:y_i=-1} w^T x_i) \tag{2.45}$$

2.4.5.3 Kernel Trick

As a linear classifier, a linearly separable dataset is usually required 2.3. But what we can do is perform a feature transform to a space where it is linearly separable.

Theorem 2. *Mercer's theorem. Let $x \in \mathbb{R}^l$ and a mapping ϕ:*

$$x \mapsto \phi(x) \in H \tag{2.46}$$

where H is a Hilbert space. The inner product operation has an equivalent representation.

$$\langle \phi(x), \phi(z) \rangle = K(x, z) \tag{2.47}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product operation in H and $K(x, z)$ is a symmetric continuous function satisfying the following conditions (known as Mercer's conditions):

$$\int_C \int_C K(x, z) g(x) g(z) dx dz \geq 0 \tag{2.48}$$

for any $g(x), x \in C \subset \mathbb{R}^l$ such that:

$$\int_C g(x)^2 dx < +\infty \tag{2.49}$$

where C is a compact (finite) subset of $\mathbb{R}^l$. [35]

From Mercer's theorem we can assume that a mapping $\phi : R^d \to R^{d'}$ from R^d to $R^{d'}$ has an equivalent kernel function K. We can now apply the mapping to the training dataset X before training begins and get a dataset X'. Then run SVM to find a separating hyperplane on the new X' dataset. We will now need to first apply the transform to any new data points, before we can make predictions on them.

Essentially we have replaced the inner product of the feature vectors with a kernel function $K(x_1, x_2) = \phi(x_1)^T \phi(x_2)$. With this in mind we can rewrite Equation 2.42:

$$\max_{a:a_i \geq 0} \sum_{i=1}^{n} a_i - 1/2 \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j y_i y_j K(x_i, x_j) \tag{2.50}$$

Subject to:

$$\sum_{i=1}^{n} a_i y_i = 0 \tag{2.51}$$

With:

$$b = -\frac{1}{2}(\min_{i:y_i=1} \sum_{a_j \neq 0} a_j y_j K(x_i, x_j) + \max_{i:y_i=-1} \sum_{a_j \neq 0} a_j y_j K(x_i, x_j))$$

$$w = \sum_{i=1}^{n} a_i y_i \phi(x_i)$$

There are many kernels that are used with SVM:

Table 2.1: Kernels used with SVM

Name	Kernel
Linear Kernel	$\langle x, x' \rangle$
Polynomial Kernel	$(\gamma \langle x, x' \rangle + r)^d$
Exponential Kernel	$exp(-\gamma \|\|x - x'\|\|^2)$
Radial Basis Function	$exp(-\gamma \|\|x—x'\|\|^2)$

Where d is the degree of the polynomial and r, γ are bias parameters.

2.4.5.4 Regularisation

For some non linearly separable datasets, the non separability might be because of a few outliers in the dataset. In this case using a feature transform might not be the best way to deal with the dataset. What we can do instead is allow a few samples to be misclassified by adding slack variables. The primal form with slack variables is defined as such:

$$\min_{w,b,\xi} 1/2||W||^2 + C \sum_{i=1}^{n} \xi_i \tag{2.52}$$

Subject to:

$$y_i(w^T x_i + b) \geq 1 - \xi_i \ \forall i \in [1, n] \tag{2.53}$$

$$\xi_i \geq 0 \ \forall i \in [1, n] \tag{2.54}$$

Where ξ_is are the slack variables, and C is the misclassification "cost".

Following the same methodology as before it is trivial to show that the problem function remains convex and the constraints linear.

The dual form of the regularized SVM problem is defined as:

$$\max_{a,r \geq 0} \min w, b, \xi L(w, b, \xi, a, r) \tag{2.55}$$

$$\max_{a,r \geq 0} \min w, b, \xi 1/2||W||^2 + C \sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} a_i(y_i(w^T x_i + b) - 1 + \xi_i) - \sum_{i=1}^{n} r_i \xi_i \tag{2.56}$$

Subject to:

$$y_i(w^T x_i + b) \geq 1 - \xi_i \ \forall i \in [1, n] \tag{2.57}$$

$$\xi_i \geq 0 \ \forall i \in [1, n] \tag{2.58}$$

From Equation 2.29 again, we know the partial derivatives of the generalized Lagrangian will be equal to zero. Taking the partial derivative of Equation 2.56 in respect to w_j:

$$\frac{\partial L(w,b,\xi,a,r)}{\partial w_j} = 0 \Rightarrow \tag{2.59}$$

$$\frac{\partial 1/2||W||^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n} a_i(y_i(w^T x_i + b) - 1 + \xi_i) - \sum_{i=1}^{n} r_i\xi_i}{\partial w_j} = 0 \Rightarrow \tag{2.60}$$

$$w_j - \sum_{i=1}^{n} a_i y_i x_{i,j} = 0 \Rightarrow \tag{2.61}$$

$$w = \sum_{i=1}^{n} a_i y_i x_{i,j} \tag{2.62}$$

So w remains unchanged.

Taking the partial derivative of Equation 2.56 with respect to b:

$$\frac{\partial L(w,b,\xi,a,r)}{\partial b} = 0 \Rightarrow \tag{2.63}$$

$$\frac{\partial 1/2||W||^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n} a_i(y_i(w^T x_i + b) - 1 + \xi_i) - \sum_{i=1}^{n} r_i\xi_i}{\partial b} = -\sum_{i=1}^{n} a_i y_i \Rightarrow \tag{2.64}$$

$$\sum_{i=1}^{n} a_i y_i = 0 \tag{2.65}$$

Again this remains unchanged.

Taking the partial derivative of Equation 2.56 in respect to ξ_i:

$$\frac{\partial L(w,b,\xi,a,r)}{\partial \xi_i} = 0 \Rightarrow \tag{2.66}$$

$$\frac{\partial 1/2||W||^2 + C\sum_{i=1}^{n}\xi_i - \sum_{i=1}^{n} a_i(y_i(w^T x_i + b) - 1 + \xi_i) - \sum_{i=1}^{n} r_i\xi_i}{\partial \xi_i} = -\sum_{i=1}^{n} a_i y_i \Rightarrow \tag{2.67}$$

$$C - a_i - r_i = 0 \Rightarrow \tag{2.68}$$

$$C = a_i + r_i \tag{2.69}$$

Now by substituting the above results into Equation 2.56 the dual regularized form can be rewritten as:

$$\max_{a,r\geq 0} \min w, b, \xi \sum_{i=1}^{n} a_i - 1/2 \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j y_i y_j x_i^T x_j + \sum_{i=1}^{n} (C - a_i - r_i)\xi_i \tag{2.70}$$

Subject to:

$$\sum_{i=1}^{n} a_i y_i = 0$$

$$C = a_i + r_i \; \forall i \in [1, n]$$

We can now note that r does not appear in the problem function and that we can always choose $r_i \geq 0$ such that $C = a_i + r_i$ as long as $a_i \leq C$. We can also note that w, b nor ξ appear. Rewriting the regularized dual form again yields:

$$\max_{a} \sum_{i=1}^{n} a_i - 1/2 \sum_{i=1}^{n} \sum_{j=1}^{n} a_i a_j y_i y_j x_i^T x_j \tag{2.71}$$

Subject to:

$$\sum_{i=1}^{n} a_i y_i = 0 \tag{2.72}$$

$$0 \leq a_i \leq C \; \forall i \in [1, n] \tag{2.73}$$

> **Note**
>
> The kernel trick from 2.4.5.3 Kernel Trick, still applies.

2.5 Sequential Minimization Optimization

The dual form we have derived in Equation 2.42, as well as the kernelized form in Equation 2.50 can be solved using standard quadratic programming (QP) solvers. Using those can be very performance and memory intensive. For that reason many techniques where developed to speed up SVM. One of the more successful ones was Sequential Minimization Optimization (SMO) which takes the relatively large QP problem of SVM, breaks it into many smaller ones, and solves them analytically [7]. For

each optimization step due to the linear constraint in Equation 2.72, two Lagrange multipliers are jointly optimized at a time.

More specifically a high level overview of how the SMO algorithm works is as follows:

1. Pick one Lagrange multiplier to a_1 optimize, using the First Choice Heuristic.
2. Pick a second Lagrange multiplier a_2 to optimize, using the Second Choice Heuristic.
3. Calculate the prediction error $E_i =$ Output of SVM on point i $- y_i$ for the first multiplier.
4. Using the prediction errors, perform what is essentially coordinate descend [36, page 11], to move both multipliers closer to their optimal value.
5. This is repeated until all multipliers satisfy the KKT conditions, within a small margin ϵ, the original paper [7, page 48, Loose KKT Conditions] recommends a value in the range of 10^{-2} to 10^{-3} for the margin.

The First Choice Heuristic attempts to find a Lagrange multiplier that violates the KKT conditions. To speed up the training multipliers that are bounded[6] are ignored –except if a full pass over the training set has not found a violating multiplier. In this case a full pass over all multipliers is done to find violating ones.

The Second Choice Heuristic attempts to find a second Lagrange multiplier, a_2, one that maximises the absolute value of the prediction error on the samples i_1 and i_2 given an already decided i_1.

[6]bounded multipliers: when $a_i \neq C,\ a_i \neq 0$

3 CUDA

Graphics Processing Units (GPU)s, as the name implies, were originally created to accelerate graphical and image processing applications. Graphics programming at it's core is an inherently parallel computing problem, thus GPUs where made to support such massively parallel computing. Some of the earliest attempts at general purpose computing on graphics processing units (GPGPU) where done using programming models intended for graphics processing which was rather cumbersome and inelegant.

Compute Unified Device Architecture, or what is more commonly known as CUDA, is Nvidia's GPGPU Application Programming Interface (API) and computing platform. It includes a C/C++ compiler, `nvcc` which is based on LLVM, and a sleuth of libraries for GPU accelerated processing: including but not limited to cuBLAS (Basic Linear Algebra Subprograms), cuFFT (Fast Fourier Transform) and more. Official Support for CUDA also exists for Fortran and unoffical support exists in other languages that support foreign function interfaces (FFI) into C code, using wrappers from third parties.

3.1 Architecture of Nvidia GPUs

Firstly, before diving into the details of the CUDA platform, we will need to define some terms and lay out the architecture of Nvidias GPUs. GPUs are multiprocessors which support running hundreds of threads at the same time by employing an architecture called Single-Instruction Multiple-Threads (SIMT). Much like traditional Single-Instruction Multiple-Data (SIMD) architectures employed by CPUs, SIMT architectures operate on multiple data with the using the same instruction, but unlike SIMD that takes advantage of vectorized instructions and registers, the execution of on instruction does not necessarily happen at the same time, but instead happens concurrently by different threads. To implement such an architecture Nvidia GPUs are made up of individual CUDA cores equipped with their own registers, L1 cache and Program Counter (PC)[1] arranged into multiple Streaming Multiprocessors (SM)s with additional shared L2 cache memory. In the context of Nvidia GPUs, the job of the multiprocessor is to create, manage the execution of and schedule groups of 32 threads, called warps, by partitioning bigger

[1] In architectures after NVIDIA Volta, where Independent Thread Scheduling was added.

groups of threads, called thread blocks. Warps execute only a single instruction at a time and as such in the case of divergence due to different code paths, threads are selectively disabled so that each group of threads with a divergent code path is executed on it's own time. While instruction level parallelism is employed by way of pipelining, all instructions are executed in order and no branch prediction being employed[2].

3.2 Programming Model

The CUDA programming model is a stream programming model where a kernel function (not to be confused with the kernel functions defined in 2.4.5.3 Kernel Trick) is applied to a stream of data points. This is an embarrassingly parallel workload that obviously maps very well with to the SIMD architecture GPUs use. It is also a heterogeneous programming model where a distinction is made between code that is to be run on the CPU, known as host code, and code that is meant to be run on the GPU, known as device code. Host code is perfectly normal C/C++ and abides by that language's syntax and rules, with the exception of kernel invocation, see 3.2.1 Execution and Threading Model. Device code on the other hand, while syntactically identical with host code, comes with some major restrictions: Big parts of the C standard library and C++'s standard template library (STL) are unavailable, with exceptions such as `printf` to facilitate the printing of debug information.. Device code also has access to a wide range of device only library functions and compiler intrinsics. A full list of the restrictions and extensions available exists in the CUDA C++ Programming Guide [37, see sections: C++ Language Extentions and C++ Language Support]

3.2.1 Execution and Threading Model

Functions can be marked as host code, device code or kernels, using the compiler attributes `__host__` for host code, `__device__` for device code and `__global__` for kernels. A function can be marked both `__host__` and `__device__`, to imply that a function can be called from both host and device code.

Kernel functions are executed in parallel N times by N cuda threads. CUDA threads are the lowest class in the threading model. At the top of the threading model hierarchy are *grids*, which map to the available hardware GPUs. Grids are then made up of *blocks* with multiple blocks being executed concurrently by one of the SMs of the GPU. Blocks are made up of individual cuda *threads* which map into individual

[2]Thankfully, the author calls upon the reader to imagine what speculative execution vulnerabilities like Meltdown/Spectre would entail on GPUs.

CUDA cores. Optionally, *blocks* can also be grouped into *thread block clusters* to guarantee that they are run on the same *grid* in multi-GPU systems. Threads are enumerated sequentially with their own thread ID, unique within each block, and block ID. While the simplest form of ID is a simple scalar, 2D or 3D vectors can be used when such shapes of blocks and grid are employed.

Kernel functions must have a return type of `void` and must be free functions and not methods of any class. Listing 3.1 contains an example of a kernel function `kernel`, marked with the `__global__` attribute. In `main()` the syntax used for kernel invocation is demonstrated. Kernel invocation is like a normal function call except that the function name is prepended with triple angle brackets `<<<...>>>` which is know as an execution configuration.

Listing 3.1 CUDA Kernel Invocation

```
__global__ void kernel() {
    printf("Hello from %d!\n", threadIdx.x);
}

int main(void) {
    kernel<<<32,32>>>();

    return 0;
}
```

The parameters given in the execution configuration are `gridDim` of type `dim3`[3] denoting the number and shape of blocks to be used, `blockDim` of type `dim3` denoting the number and shape of threads to be used by each block, `shared_size` of type `size_t`, an optional parameter, denoting the size of dynamically allocated shared memory for each block and `stream` an optional parameter defaulting to `0` denoting the id of the stream the kernel will use.

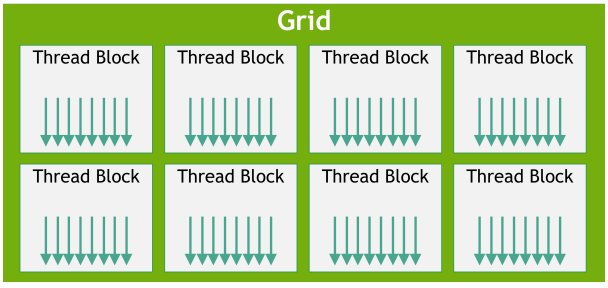

Figure 3.2.1: Grid of Thread Blocks

Builtin variables are provided in device code so that individual threads can use them to index into and operate on different data:

[3]dim3: a struct containing the atributes `x`, `y` and `z` of integer type describing a three dimentional vector, can be initiallized with a single integer to imply a vector with y=1 and z=1

- gridDim, of type `dim3`, signifies the shape of the grid
- blockDim, of type `dim3`, signifies the shape of each block
- blockIdx, of type `dim3`, signifies the ID of each block
- threadIdx, of type `dim3`, signifies the ID of each thread
- warpSize, of type `int`, signifies the size of the warp

Using these builtins, a simple addition between two vectors could be implemented as in Listing 3.2, where `SIZE_OF_VECTORS` is smaller than the total number of threads using in the execution configuration for this kernel.

Listing 3.2 CUDA Addition Kernel

```
__global__ void add(double *a, double *b, double *result) {
    unsigned int tid = threadIdx.x + blockDim.x * blockIdx.x;

    if (tid < SIZE_OF_VECTORS) {
        result[tid] = a[tid] + b[tid];
    }
}
```

3.2.1.1 Synchronization

Synchronization between threads is achieved using compiler intrinsics. Block wide synchronization, as in synchronization between threads in the same block, is provided using the `__syncthreads()` primitive, which acts like a barrier where all threads in a block must wait until all of them reach it.

> **Warning**
>
> It's important to note that if one thread in a block reaches a barrier, all threads must eventually reach it, otherwise a deadlock will happen. Extra caution must be taken when placing `__syncthreads()` in conditional branches.

Warp wide synchronization and cooperation is achieved using:

- warp vote intrinsics which implement warp wide reduce and broadcast operations [37, section Warp Vote Functions]
- warp reduce intrinsics which implement warp wide reductions [37, section Warp Reduce Functions]

- warp shuffle intrinsics which implement ways for threads exchange variables in a warp without using any shared memory [37, section Warp Shuffle Functions]

Grid synchronization intrinsics are not provided and as such has been achieved in many different ways, one of them being by using multiple kernel invocations. Synchronizations happened at the end of each kernel invocation.

3.2.1.2 Cooperative Kernels

Introduced in CUDA 9, cooperative kernels allowed more granular control of synchronization, including and not limited to grid wide synchronization, warp barrier synchronization and synchronization between specific groups of blocks. Using this cooperative model is done by including the `cooperative_groups.h` header file. Kernel invocation for cooperative kernels differs as it must make use of the `cudaLaunchCooperativeKernel` API, instead of execution configuration as mentioned before.

Synchronization is achieved using barriers on groups of threads. There are many predefined kinds of thread groups including but not limited to: *thread block* groups, corresponding to the traditional thread groups synchronized by `__syncthreads()`, *grid* groups, corresponding to an entire grid of threads and more [37, section Cooperative Groups].

3.2.2 Memory

Memory in CUDA is separated in many different categories, the first one being host and device memory. Host memory is inaccessible from device code and vise versa. Data must be explicitly moved from the host to the device and back by using one of the variants of `cudaMemcpy*(src, dst, size, kind)`, where kind defines if the `src` and `dst` are on the host or device side. Device to device and host to host copying is also supported.

On the device side memory is of four kinds:

- *Global* memory, very slow memory that is accessible to the entire grid
- *Block* shared memory, fast memory that is shared across a block, maps to the memory of each SM.
- *Local* thread memory, very fast memory that is local to each thread, maps to the cache and registers of individual CUDA cores.
- *Constant* memory, fast globally accessible memory that is however, immutable

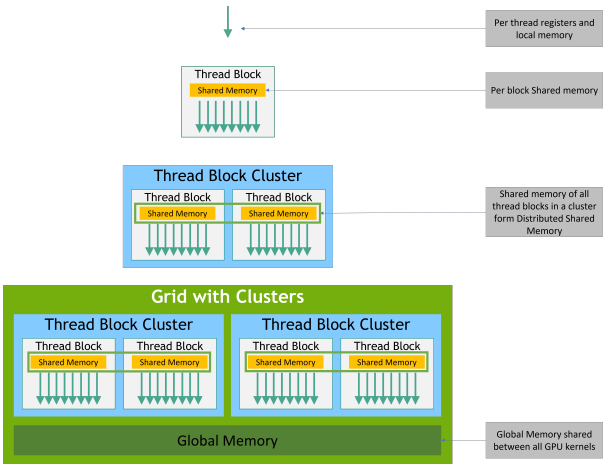

Figure 3.2.2: Memory Hierarchy

3.2.2.1 Unified Memory

Unified or managed memory, introduced with CUDA, 6 allows the programmer to act as if they operate on a unified memory space by automagically managing the transfer of data from host to device and back. This can massively simplify the development of an application by removing the need to manually manage transfer of data. Unified memory must be allocated using the `cudaMallocManaged()` memory allocation API [37, section Unified Memory Programming], because it requires the use of page-locked or pinned memory. Pinned memory is simply memory, as far as the host side is concerned, is memory that cannot be stored with the use secondary storage such as swap memory on *nix systems or pagefiles on Windows systems.

3.2.3 Compilation

Device code is written in the CUDA instruction set architecture known as `PTX`, directly using it is as cumbersome as writing `x86_64` assembly is, so as mentioned before, device code is written in an extended syntax of C/C++. Compilation of host and device code is done using split compilation by first separating host from device code, with the compiler `nvcc` compiling the device code into `PTX` code itself and handing off the host code to the system's C compiler (usually `gcc` or `MSVC`). During runtime `PTX` code is Just In Time (JIT) compiled to machine code and cached for future use.

> **Tip**
>
> `clang` can also be used to compile CUDA code [38], but it uses what is known as *merged parsing*, the clang documentation claims that "[…] clang's approach allows it to be highly robust to C++

> edge cases, as it doesn’t need to decide at an early stage which declarations to keep and which to throw away […]” [39].

4 Parallelization of SVM

The use of parallelization techniques for speeding up SVM has been the topic of a lot of many research papers. This is because training SVM classifiers is a very performance and memory intensive task that has lots of potential to make use of parallelization. A lot of the research is focused on GPGPU solutions, this is due to the high floating point performance of GPUs and their price to performance ratio compared to other techniques that require entire clusters of hardware to work. This section will cover the use of both traditional parallelization techniques as well as CUDA based GPGPU techniques and the algorithms that have been developed using them.

4.1 Parallelization Techniques

There many parallelization techniques that show up in the available body of research. One of them being the parallelization of the SVM algorithm itself. Another one is the division of big training sets into smaller parts and then training SVMs on those in parallel with the goal of having a smaller working set of the dataset in memory. Another technique employed has been parallel grid search, where multiple version of the same model are trained side by side, with different hyperparametes, in order to find the optimal hyperparametes for a given dataset.

4.2 Parallel SVM Algorithms:

4.3 P-SMO

Parallel Sequential Minimization Optimization or P-SMO, is an improvement of the SMO algorithm described in section 2.5. It attempts to break the dataset into N parts and then assign each part to one of N processors in an effort to minimize training and prediction times.

The main idea of P-SMO, breaking the dataset into many smaller parts, is based on separating the dataset into part depending on each sample's Lagrangian multiplier and the sign of it's class. More specifically they define for each processor k:

$$I_0^k = \{i : y_i = 1,\ 0 < a_i < C\} \cup \{i : y_i = -1,\ 0 < a_i < C\} \tag{4.1}$$

$$I_1^k = \{i : y_i = 1,\ a_i = 0\} \tag{4.2}$$

$$I_2^k = \{i : y_i = -1, a_i = C\} \tag{4.3}$$

$$I_3^k = \{i : y_i = 1, a_i = C\} \tag{4.4}$$

$$I_4^k = \{i : y_i = -1, a_i = 0\} \tag{4.5}$$

Where C is the hyperparameter representing the cost of misclassification.

With I^k for all processors, signifying the all indexes of the dataset the processor k has been assigned.

They define then define two bias terms b_{low} and b_{up} instead of just one:

$$\begin{aligned} b_{up}^k &= \min\{E_i : i \in I_0 \cup I_1 \cup I_2\} \\ b_{low}^k &= \max\{E_i : i \in I_0 \cup I_3 \cup I_4\} \\ &\text{where } E_i \text{ is the prediction error on sample } i \end{aligned} \tag{4.6}$$

And their associated indices as:

$$\begin{aligned} I_{up}^k &= \text{argmin} E_i \\ I_{low}^k &= \text{argmax} E_i \end{aligned} \tag{4.7}$$

Then they match each index, up and low, to one of the two Lagrangian multipliers a_1 and a_2 as defined in the SMO algorithm in section 2.5, without loss of generality assume that $a_1 = a_{up}^k$ and $a_2 = a_{low}^k$:

> **Note**
>
> They define the prediction error as $E_i^k = \sum_{j=1}^{l} a_j y_j K(x_j, x_i) - y_i$, as expected.

For each processor k:

$$a_{I_{low}}^{new} = a_{I_{low}}^{o} ld - \frac{y_2(E_{I_{low}}^{old} - E_{low}^{old})}{\eta} \tag{4.8}$$

$$a_{I_{up}}^{new} = a_{I_{up}}^{old} + (y_{low} y_{up})(a_{I_{low}}^{old} - a_{I_{low}}^{n} ew) \tag{4.9}$$

Where $\eta = 2K(x_1, x_2) - K(x_1, x_1) - K(x_2, x_2)$ and $K(...)$ is the kernel function. Also, again as in SMO, both a_{low} and a_{up} are clipped to $(0, C)$.

For the stopping criteria they define they the duality gap as the distance between the primal and the dual objective function and the dual value (which is updated at each step) as:

$$\text{dual}^{new} = \text{dual}^{old} - \frac{a_{I_{up}}^{new} - a_{I_{up}}^{old}}{y_i}(E_{I_{up}}^{old} - E_{I_{low}}^{old}) + {}^{1}\!/_{2}\eta(\frac{a_{I_{up}}^{new} - a_{I_{up}}^{old}}{y_i})^2 \tag{4.10}$$

$$\text{duality gap}^{k} = \sum_{i=0}^{l} a_i y_i E_i + \sum_{i=0}^{l} \epsilon_i \tag{4.11}$$

Where each processor p calculates it's own **duality gap** and the final value is given by summing all the values from each processor:

$$\text{duality gap} \sum_{p=1}^{\text{v}} \text{duality gap}^{k} \tag{4.12}$$

Where v is the total number of cpus.

The stop criteria is hit when the duality gap is smaller or equal to the absolute value of the dual value times a constant $\tau = 10^{-6}$.

$$\text{duality gap} \leq \tau |dual| \tag{4.13}$$

The pseudocode for the algorithm is in Listing 4.1:

4.3.1 Results

The implementation of P-SMO was done using the MPI (Message Passing Interface) library, a parallel/distributed computing library available for C/C++ and Fortran. Testing was done on an IBM p690 Regata SuperComputer with a total of 7 nodes, each with 32 Power_PC4 1.3Ghz cores. Experiments

Listing 4.1 P-SMO Pseudocode

```
for all p procesors
    init a[i] = 0
    init Error[i] = - y[i]
    init gap = 0
done

while gap < tau * |dual| each processor
    optimize a[I_up], a[I_low]
    update E_i for all indices assigned to processor
    calculate b_up, b_low, I_up, I_low and gap of each processor
    reduce and broadcast b_up, b_low, I_up, I_low and gap
end
```

presented by Cao, Keerthi, Ong, *et al.* [8] in their paper show a sizable speedup when compared to both their own sequential SMO [7] implementation and state of the art LIBSVM [40] while maintaining high prediction accuracy.

4.4 Parallel-Parallel SMO

Parallel-Parallel SMO or P2SMO is a P-SMO based GPU accelerated multiclass SVM solver. It takes advantage of the grid structure offered by CUDA to train N binary SVM classifiers, with P subsets of the dataset, in parallel by using PxN blocks of threads. By training N binary classifiers they can implement the OVA multiclass classification strategy.

Herrero-Lopez, Williams, and Sanchez [41] show further speedup can be achieved by taking advantage of the unique implications of the parallel execution. Firstly they employ cross-task caching of kernel evaluations. More specifically kernel evaluations are shared across the N different classifiers for samples that reside in the subset of the dataset split into P parts. Secondly to minimize the unnecessary launch of grids with many idle rows of blocks, due to differing convergence rates of the binary classifiers, they reduce the number of rows of each grid launched, dynamically, as classifiers reach convergence. Lastly, inference is also done in parallel by reframing the prediction function as a matrix multiplication between a matrix X, that contains the training data, and a vector z that contains the sample to be classified. The matrix multiplication was done using a standard CUBLAS function.

4.4.1 Results

Two systems were used to obtain performance metrics, one equipped with a GeForce 8800 GT and one equipped with a Tesla C1060. A speedup on the order of 3-112 times was achieved for inference and a speedup of 3-57 times was achieved for training all while maintaining high prediction accuracy.

4.5 GPUSVM

Graphics Processing Unit Support Vector Machine (GPUSVM) is a CUDA based SVM package including a training tool, a cross validation tool and a prediction tool. In this paper "GPUSVM" is going to be used to refer to the underlying algorithm of the package. It is based on P-SMO, seen in section 4.3, but adapted to run in a heterogeneous environment making use of both a GPU and CPU.

The algorithm is modified so that kernel evaluations, the computing of b_{low}^k and b_{up}^k and the optimization of the Lagrange multipliers $a_{I_{low}}$ and $a_{I_{up}}$ are all done on the GPU. In each iteration all the resulting b_{low}^k, b_{up}^k are moved to the host and reduced to the final b_{low} and b_{up} which are then used in the next iteration. The outer loop of the algorithm is run on the host, with only the inner loop running on the GPU —this is also how b_{low} and b_{up} are supplied to the GPU, by argument, to the CUDA kernel of the inner loop. In Listing 4.2 we see the pseudocode for GPUSVM.

Listing 4.2 GPUSVM Pseudocode

```
(device) init a[i] = 0
(device) init Error[i] = - y[i]
(device) init gap = 0

(host)   while gap < tau * |dual|
(device)     compute K(I_lo,I_up), K(I_up,I,up), K(I_low,I_low)
(device)     optimize a[I_up], a[I_low]
(device)     compute b_up^p, b_low^p, I_up^p, I_low^p
(host)       compute b_up, b_low, I_up, I_low
         end
```

4.5.1 Results

Being a comprehensive package, it makes training SVM models very easy for end users through the use of the supplied GUI. Testing was done by Li, Salman, Test, *et al.* [42] on a system with two Intel Xeon X680 3.3GHz 6 core CPUs, 96GBs of DDR3 1333MHz ECC RAM, six Tesla C2050s with 3GBs GDDR5 of VRAM and two Tesla C2070s with 6GBs GDDR5 of VRAM. As far as training and inference

performance, they demonstrated a quite notable speedup compared to state of the art CPU based SVM solvers such as LIBSVM [40] while maintaining high prediction accuracy.

4.6 PCV

Continued research on GPUSVM lead to the creation of the Parallel Cross-Validation algorithm (PCV), a parallel SVM solver that implements efficient multitask cross-validation. Cross-validation a technique used to find the optimal hyperparameters, such as the misclassification cost C or the degree of a polynomial kernel, to be used with a specific dataset. The idea behind the technique, dubbed n-fold cross-validation, is that to find the optimal hyperparameters for the model, the dataset can be split into n parts with each part being used as the training set and the rest as a testing set. For each fold a different subset is used as the training set. At the end of the n folds, the hyperparameters of the model with the best accuracy on the testing sets are selected. PCV runs each task with different hyperparameters in parallel so that the kernel computations as well as the data used in each fold can be shared between tasks. Kernel computations are stored in a cache that is several times smaller than would be needed to store all needed kernel computations, as such a strategy of evicting the least recently used computation is used by way of a Least Recently Used (LRU) list.

4.6.1 Results

Experiments were done on the same system as mentioned in subsection 4.5.1. A massive decrease in the total number of kernel computations was observed when compared to the previous GPUSVM while maintaining the same prediction accuracy. This resulted in an even better speedup than before when compared to LIBSVM, again while preserving a high prediction accuracy.

4.7 SVM-SMO-SDG

SVM-SMO-SDG is a hybrid of P-SMO and Stochastic Gradient Descend (SDG) used to implement an efficient data parallel SVM solver for use in a heterogeneous computing environment. The advantage of using the SDG algorithm is by speeding up the optimization of the Lagrangian Multipliers a_1, a_2 by quickly computing a new weight vector with Equation 4.14 and subsequently obtaining the value for b using Equation 4.15 and the prediction error for the samples x_i and x_j

$$w \leftarrow w - \gamma_t \begin{cases} \lambda_w, & \text{if } y_t w^T \phi(x_t) > 1 \\ \lambda_w - y_t \phi(x_t), & \text{otherwise} \end{cases} \tag{4.14}$$

$$b = y - w, x \tag{4.15}$$

4.7.1 Results

Experiments were carried out on a system equipped with a dual-core Intel Xeon CPU @ 2.20 GHz with 12GBs of RAM and an NVIDIA Tesla V100 SXM2 with 16GBs of VRAM. They concluded that the use of SVM-SMO-SDG resulted in further speedups and a significant decrease in memory usage, all while maintaining prediction accuracy.

> **Note**
>
> SVM-SMO-SDG also maintained a comparable number of support vectors produced to SMO, as opposed to PCV that produced significantly more.

4.8 C-SVM

Cascade Support Vector Machines of C-SVMs, first developed by Graf, Cosatto, Bottou, *et al.* [43], employ the second kind of parallelization mentioned in the intro of this chapter, that is, they partition the dataset into subsets and using layers of SVMs, they extract the support vectors that get passed onto the next layer. Essentially each layer of the network acts like a filter that separates important support vectors from useless data points with only the most important support vectors remaining at the end. A formal proof of convergence exists on the original paper introducing C-SVMs [43]. Intuitively, data points of a subset that exist in the margin between two classes, are likely to also exist close to the margin of the entire dataset. Also the converse must also hold true, with non-support vectors found in a subset of the dataset also not being support vectors of the entire dataset.

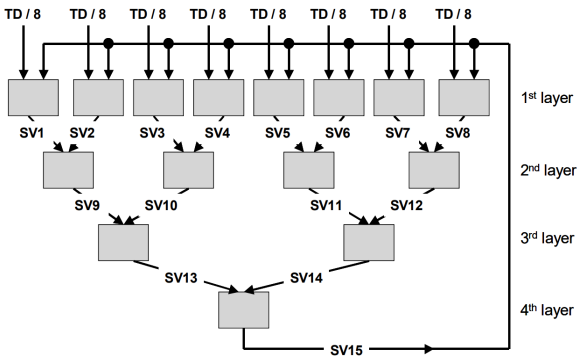

Figure 4.8.1: Architecture of a binary Cascade SVM [43]

4.8.1 Results

Experiments where performed on a system with a single processor as well as on a cluster of 16 machines, each equipped with a dual-core AMD 1800 and 2GBs of RAM. Kernel evaluations saw a significant decrease of as much as 30%. A speedup of five to up to ten times was observed as well as improvements in storage requirements. The accuracy of the resulting models was equivalent or better to that of traditional SVMs, after convergence was achieved, although satisfactory accuracy could be obtained with even a single pass through the network.

4.9 ECM

Extreme Cascade Machines (ECM)s is an extension of C-SVMs where Dimensionality Reduction (DR) is employed in order to reduce the computation requirements of the training. The use of DR is aimed at reducing the number of features used for training to speedup training and potentially improve accuracy.

4.9.1 Results

Experiments were performed using both Principal Component Analysis (PCA), ISOMAP and Locally Linear Embedding (LLE) for DR. PCA-SVM showed the greatest speedup compared to C-SVM, as well as higher accuracy, even succeeding in training where C-SVM did not successfully complete.

5 Implementation

This section will cover two implementations, the serial SMO SVM solver meant to server as a baseline and a parallel GPUSVM-based SVM solver. The implementation was written in C++17 and CUDA 12[1].

5.1 CLI Interface

The usage of both implementations is done through a Command Line Interface (CLI). The resulting binary can be run as is, to run with the default settings, or it can accept various flags and options.

> **Note**
>
> Parsing of the cli arguments was done with the use of the header-only argparse library [44].

It accepts two positional arguments: `DATASET`, which is the name of the dataset to use, and `ALGO` which indicates whether to run the CPU (Central Processing Unit) algorithm (SMO) or the GPU algorithm (GPUSVM). It accepts three optional arguments, `--threads <integer>`, which controls the number of CUDA threads to be used for the GPU algorithm, `--blocks <integer>`, which controls the number of CUDA blocks to be used, and lastly `--test` which runs a test of the model on the training data, to obtain a prediction accuracy. The usage of the binary, accessible through the flag `--help`, follows in Listing 5.1.

[1] exact cuda version: Cuda compilation tools, release 12.3, V12.3.107 Build cuda_12.3.r12.3/compiler.33567101_0

Listing 5.1 Usage of the cli interface

```
Usage: svm [--help] [--version] [--blocks VAR]
           [--threads VAR] [--size VAR] [--test] DATASET ALGO

Positional arguments:
  DATASET        the dataset to use [nargs=0..1] [default: "linear"]
  ALGO           algorithm to use [nargs=0..1] [default: "cpu"]

Optional arguments:
  -h, --help     shows help message and exits
  -v, --version  prints version information and exits
  -b, --blocks   number of blocks for CUDA [nargs=0..1] [default: 16]
  --threads      number of threads for CUDA [nargs=0..1] [default: 128]
  --size         size of the linear DATASET [nargs=0..1] [default: 1000]
  --test         test the model after training
```

5.2 Serial SMO

The serial implementation is meant to serve as a baseline for benchmarking so it's a faithful implementation of SMO [7]. In Listing 7.8 we see the outer loop of the SMO algorithm, the outer loop keeps running until we have examined all examples and changed no multipliers thus made no further progress. In the case where no progress has been made when checking non-bound multipliers, the entire set of multipliers is checked before giving up. An iteration limit has also been used to stop training if it has been stuck slowly optimizing a few multipliers. The inner loop in Listing 7.9 implements the second order choice heuristics The `takeStep()` function in Listing 7.10 implements the optimization step of SMO. In the case of a negative η, the chosen multiplier is skipped instead of evaluating the objective function at L and H because in experiments it resulted in reaching the iteration limit due to the algorithm being stuck when a negative η was common.

5.3 Parallel GPUSVM

The parallel implementation is heavily based on GPUSVM [42] and P-SMO [8]. The main difference when compared to GPUSVM is the use of cooperative kernels and grid synchronization in order to eliminate the need to move data from the host to the device and back at each iteration. Instead a grid wide reduction is implemented to obtain b_{low} and b_{up}, seen in Listing 7.11. The reduction implements the argmin and argmax operation at the same time, returning two results. In the first stage each CUDA thread finds a local max and min, as well as their indices. Next a block reduction is performed to obtain block local results which are then written to global device memory by each thread with `threadIdx.x == 0`. In the final stage a grid synchronization is performed and then the first block, the one with `blockIdx.x`

`== 0`, performs a block reduction of the previously mentioned block local results and the results are written to global memory. Afterwards a grid synchronization is again performed and all threads read the results into their local memory.

> **Warning**
>
> Attempting to run with too many threads, will cause the occupancy routine to return 0, so running will be canceled with an exit status of 1, and an appropriate error message.

Because of the use of cooperative kernels, the optimal number of blocks to use can actually be very easily determined, dynamically. In Listing 5.2 we see the code used to achieve the optimal SM occupancy, the code is also available on the official CUDA documentation [37]. This is done by querying the platform for the maximum number of active blocks per SM that can be used with a specific kernel and a specific number of threads.

Listing 5.2 Calculation of optimal number of blocks

```
/// This will launch a grid that can maximally fill the GPU, on the default stream with kernel arguments
int numBlocksPerSm = 0;
// Number of threads my_kernel will be launched with
cudaGetDeviceProperties(&deviceProp, dev);
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocksPerSm, train_CUDA_model, THREADS, 0);
dim3 dimBlock(THREADS, 1, 1);
dim3 dimGrid(deviceProp.multiProcessorCount * numBlocksPerSm, 1, 1);
```

5.4 Vector Library

In order to better abstract the algorithms as well as to provide code deduplication and promote code reuse between the two implementations, a thin wrapper library over raw C arrays was developed. The library provides generic statically sized vector and matrix types with device and host variants for easy use in both normal C++ code, and CUDA code. Each type resides in their own header files, but the matrix header depends on the vector header. To facilitate further abstraction over raw arrays and avoid error prone manual data moves between host and device, constructors have been implemented that convert from host vectors to CUDA vectors and vice-versa. A useful functional-style `mutate()` method is provided for the vector type, one which accepts a lambda and applies it to each element of the vector, this enabled an easy and less error-prone way to reason about the data contained vectors. The source code for the vector and matrix types is provided in the appendix in Listing 7.1 and Listing 7.5.

6 Experimental Results

This section will go over the experiments and the results obtained as well as the datasets used to perform the experiments.

6.1 Hardware

Experiments were done on 2 systems, a personal workstation with a WSL (Windows Subsystem for Linux) based VM on a Windows 11 host, referred to as “WSL” from this point on, and a dedicated headless GNU/Linux system provided by the university, referred to as “Headless” from this point on. The hardware specifications for the systems follow in Table 6.1 and Table 6.2.

Table 6.1: WSL System

Component	Description
CPU	6-core AMD Ryzen 5 3600 @ 3.60GHz
RAM	24 GB (12GiB allocated to VM)
GPU	Nvidia GeForce 1060
VRAM	6GB

Table 6.2: Headless System

Component	Description
CPU	8-core AMD Ryzen 7 3700X @ 3.60GHz
RAM	66 GB
GPU	Nvidia TITAN RTX
VRAM	24GB

6.2 Datasets

6.2.1 Linear

A synthetic, linearly separable dataset of various sizes, from one thousand up to ten million, was generated using the python script in Listing 7.16. In all experiments the accuracy rate stayed above 98%, so no interesting comparisons are there to be made regarding accuracy. This dataset will be referred to as the “linear N”,dataset past this point where N is the number of samples .

6.2.2 Iris

The iris dataset consists of 150 samples of 4 features of 3 types of iris plants, Iris Setosa, Iris Versicolor and Iris Virginica. The features are describing the sepal length, sepal width, petal length and petal width. Each class is represented by 50 samples. One of the classes is linearly separable in respect to the other two, but the other two are not linearly separable in respect to each other [45]. Due to the separability discussed above, the accuracy of our classifier was low, as to be expected since it’s only a linear classifier.

6.3 Baseline SMO vs GPUSVM

All sizes of the linear dataset were used to compare the sequential and parallel implementations. In Figure 6.3.1 we can clearly see that the parallel implementation is several orders of magnitude faster than the sequential implementation, as long as the dataset is big enough. For small datasets the overhead of the parallel implementation should be big enough that the sequential implementation is expected completes training faster. It’s important to note that results using the sequential implementation and a dataset size of 100000 (hundred thousand) and more do not exist, as the runtime to was prohibitively large, as shown by the regression represented by the dotted line. The data used can be seen in Table 7.1.

6.4 Varying Number of Threads

Experiments were done using a varying numbers of threads on the linear dataset with 10 million samples. Recall that due to the use of cooperative kernels, the optimal number of blocks depends on the number of threads and the GPU installed in the system. Given that number of threads must be a power of two and such that `blockDim.x >= gridDim.x`, or more clearly such that there are more threads per block than blocks, in order for the parallel grid reduction to function correctly, the number of threads experimented

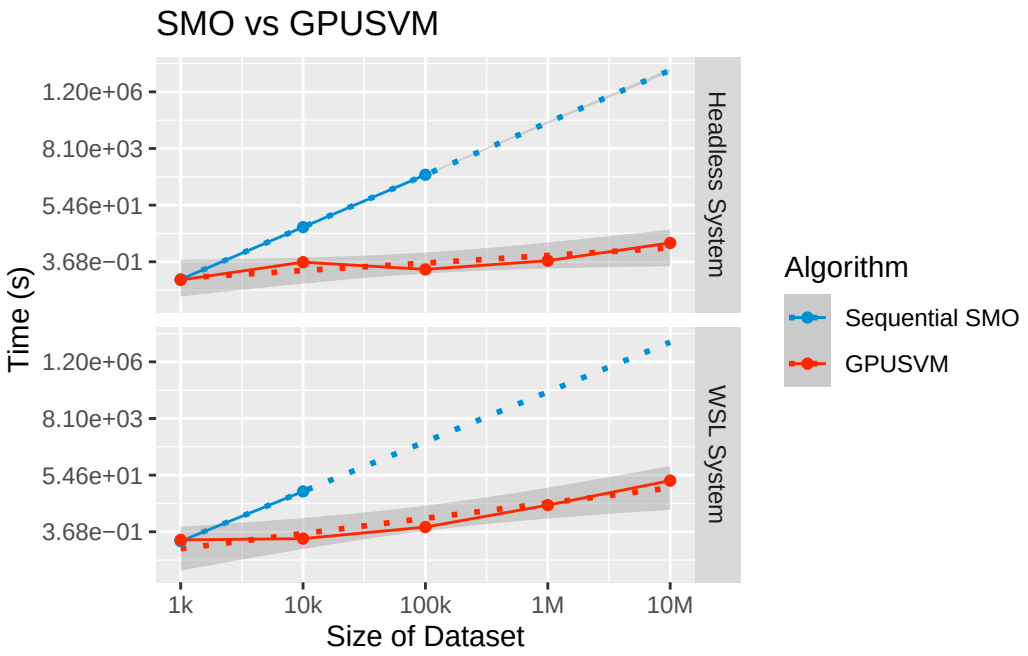

Figure 6.3.1: Training time of serial SMO and parallel GPUSVM

with were picked such that the above hold true. In Figure 6.4.1, we can see that the number of threads can have a significant effect on the training time, with one experiment resulting in a 20% slowdown compared to the rest. The data used can be seen in Table 7.2.

6.5 Small Datasets

For small datasets, as seen before, we would expect the overhead of GPUSVM to actually introduce a significant slowdown when compared to Sequential SVM. But, as it turns out this is not always true. In Figure 6.5.1 we see that if the GPU is powerful enough, the parallel implementation can actually compete, even for small datasets, as show by the data for the Linear 1k dataset on the Headless system equipped with the Nvidia TITAN RTX. The data used can be seen in Table 7.3.

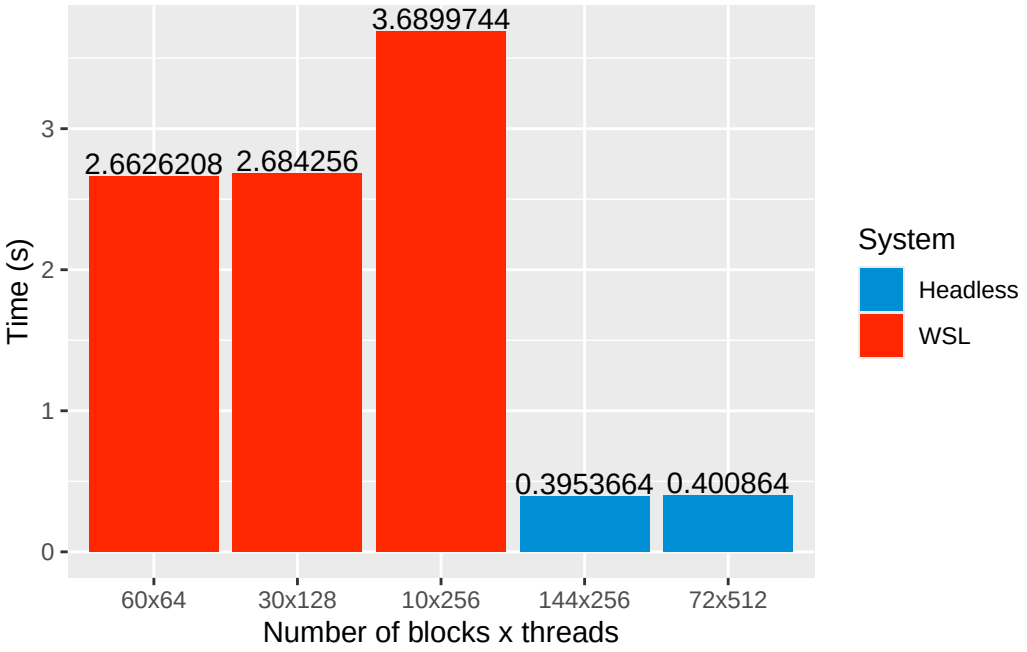

Figure 6.4.1: Training time for differing thread count

Figure 6.5.1: Training time for small datasets by algorithm

7 Conclusions and Future work

As seen from our experimental results the implementation of an efficient parallel classifier using CUDA can result in significant training time decrease when compared to sequential implementations. Our implementation was successful in leveraging the compute of the GPU of the systems we had available, but the implementation did not fully take advantage of the heterogeneous computing environment, that is to say, the CPU remained largely unused while the GPU did work. Furthermore implementation of different kernels other than the linear, dot product, kernel was unsuccessful. We also showed that the hardware used has a significant effect on the training time.

Work was attempted in order to consider the performance impact of different floating point precisions, but changing the floating point precision from *double* to *single* was harder than expected, with *half* precision requiring major rework of the code base.

Future work should consider the performance impact that floating point precision, be it single, double, half or even quarter could have. It could lead to a significant speedup when training but caution should be exercised that the accuracy of the model is not degraded. Another interesting feature of a parallel SVM solver would be cooperative kernel evaluation, where threads cooperate to find kernel values, which could have a significant impact on the training speed of datasets with high dimensionality. Lastly fully taking advantage of a heterogeneous environment, implementing a hybrid SVM solver that can run on both the GPU and CPU, should be considered.

With this and the massive body of research already available on the subject of GPGPU in mind, the benefits of use of GPUs for general purpose computing is evident. Leveraging GPUs for accelerating the training of SVMs can induce a massive speedup on training time.

References

[1] R Core Team, *R: A language and environment for statistical computing*, R Foundation for Statistical Computing, Vienna, Austria, 2023. [Online]. Available: https://www.R-project.org/.

[2] H. Wickham, D. Vaughan, and M. Girlich, *Tidyr: Tidy messy data*, R package version 1.3.1, 2024. [Online]. Available: https://CRAN.R-project.org/package=tidyr.

[3] H. Wickham, *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016, ISBN: 978-3-319-24277-4. [Online]. Available: https://ggplot2.tidyverse.org.

[4] H. Wickham, T. L. Pedersen, and D. Seidel, *Scales: Scale functions for visualization*, R package version 1.3.0, 2023. [Online]. Available: https://CRAN.R-project.org/package=scales.

[5] J. Allaire, C. Teague, C. Scheidegger, Y. Xie, and C. Dervieux, *Quarto*, version 1.4, Feb. 2024. DOI: 10.5281/zenodo.5960048. [Online]. Available: https://github.com/quarto-dev/quarto-cli.

[6] C. Cortes and V. N. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, pp. 273–297, 2004.

[7] J. Platt, "Sequential minimal optimization: A fast algorithm for training support vector machines," Microsoft, Tech. Rep. MSR-TR-98-14, Apr. 1998. [Online]. Available: https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fast-algorithm-for-training-support-vector-machines/.

[8] L. J. Cao, S. S. Keerthi, C. J. Ong, *et al.*, "Parallel sequential minimal optimization for the training of support vector machines," *IEEE Trans. Neural Networks*, vol. 17, no. 4, pp. 1039–1049, 2006.

[9] N. A. Mahoto, A. Shaikh, A. Sulaiman, M. S. A. Reshan, A. Rajab, and K. Rajab, "A machine learning based data modeling for medical diagnosis," *Biomedical Signal Processing and Control*, vol. 81, p. 104 481, 2023, ISSN: 1746-8094. DOI: https://doi.org/10.1016/j.bspc.2022.104481. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1746809422009351.

[10] K. Kowsari, K. Jafari Meimandi, M. Heidarysafa, S. Mendu, L. Barnes, and D. Brown, "Text classification algorithms: A survey," *Information*, vol. 10, no. 4, 2019, ISSN: 2078-2489. DOI: 10.3390/info10040150. [Online]. Available: https://www.mdpi.com/2078-2489/10/4/150.

[11] A. Μάργαρης, "Identification of buildings from multimodal spatial data using machine learning techniques," University of West Attica, Mar. 2023. DOI: http://dx.doi.org/10.26265/polynoe-3743.

[12] E. Ngai, Y. Hu, Y. Wong, Y. Chen, and X. Sun, "The application of data mining techniques in financial fraud detection: A classification framework and an academic review of literature," *Decision Support Systems*, vol. 50, no. 3, pp. 559–569, 2011, On quantitative methods for detection of financial fraud, ISSN: 0167-9236. DOI: https://doi.org/10.1016/j.dss.2010.08.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167923610001302.

[13] P. Cunningham, M. Cord, and S. J. Delany, "Supervised learning," in *Machine Learning Techniques for Multimedia: Case Studies on Organization and Retrieval*, M. Cord and P. Cunningham, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 21–49, ISBN: 978-3-540-75171-7. DOI: 10.1007/978-3-540-75171-7_2. [Online]. Available: https://doi.org/10.1007/978-3-540-75171-7_2.

[14] P. Dayan, M. Sahani, and G. Deback, "Unsupervised learning," *The MIT encyclopedia of the cognitive sciences*, pp. 857–859, 1999.

[15] S. Suthaharan, "Big data classification: Problems and challenges in network intrusion prediction with machine learning," *SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 4, pp. 70–73, Apr. 2014, ISSN: 0163-5999. DOI: 10.1145/2627534.2627557. [Online]. Available: https://doi.org/10.1145/2627534.2627557.

[16] Y. Liang, *Cos 495: Machine learning basics lecture 3: Perceptron*, Princeton University. [Online]. Available: https://www.cs.princeton.edu/courses/archive/spring16/cos495/slides/ML_basics_lecture3_Perceptron.pdf.

[17] S. C. Hoi, D. Sahoo, J. Lu, and P. Zhao, "Online learning: A comprehensive survey," *Neurocomputing*, vol. 459, pp. 249–289, 2021, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2021.04.112. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231221006706.

[18] B. Liang, T. Wang, S. Li, W. Chen, H. Li, and K. Lei, "Online learning for accurate real-time map matching," in *Advances in Knowledge Discovery and Data Mining*, Springer International Publishing, 2016, pp. 67–78. DOI: 10.1007/978-3-319-31750-2_6. [Online]. Available: https://doi.org/10.1007%2F978-3-319-31750-2_6.

[19] S. Gupta and N. Mehra, "Survey on multiclass classification methods," Jul. 2013.

[20] W. Mcculloch and W. Pitts, "A logical calculus of ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, pp. 127–147, 1943.

[21] F. Rosenbaltt, “The perceptron–a perciving and recognizing automation,” *Cornell Aeronautical Laboratory*, 1957.

[22] M. Minsky and S. Papert, “An introduction to computational geometry,” *Cambridge tiass., HIT*, vol. 479, no. 480, p. 104, 1969.

[23] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (Springer series in statistics). Springer, 2001, pp. 349–350, ISBN: 9780387952840. [Online]. Available: https://books.google.gr/books?id=VRzITwgNV2UC.

[24] K. Gurney, *An introduction to neural networks*. CRC press, 1997.

[25] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[26] A. Ajit, K. Acharya, and A. Samanta, “A review of convolutional neural networks,” in *2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, 2020, pp. 1–5. DOI: 10.1109/ic-ETITE47903.2020.049.

[27] J. Schmidhuber, *Annotated history of modern ai and deep learning*, 2022. arXiv: 2212.11279 [cs.NE].

[28] T. Cover and P. Hart, “Nearest neighbor pattern classification,” *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967. DOI: 10.1109/TIT.1967.1053964.

[29] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Commun. ACM*, vol. 18, no. 9, pp. 509–517, Sep. 1975, ISSN: 0001-0782. DOI: 10.1145/361002.361007. [Online]. Available: https://doi.org/10.1145/361002.361007.

[30] S. M. Omohundro, *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[31] J. MacQueen *et al.*, “Some methods for classification and analysis of multivariate observations,” in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Oakland, CA, USA, vol. 1, 1967, pp. 281–297.

[32] V. I. Levenshtein, “Binary Codes Capable of Correcting Deletions, Insertions and Reversals,” *Soviet Physics Doklady*, vol. 10, p. 707, Feb. 1966.

[33] H. Zhang, “The optimality of naive bayes,” *Aa*, vol. 1, no. 2, p. 3, 2004.

[34] G. Gordon and R. Tibshirani, *Karush-kuhn-tucker conditions, optimization*. [Online]. Available: http://www.cs.cmu.edu/~ggordon/10725-F12/slides/16-kkt.pdf.

[35] S. Theodoridis and K. Koutroumbas, *Pattern Recognition*. Jan. 2003, ISBN: 0-12-685875-6.

[36] S. J. Wright, “Coordinate descent algorithms,” *Mathematical Programming*, vol. 151, no. 1, pp. 3–34, Jun. 2015, ISSN: 1436-4646. DOI: 10.1007/s10107-015-0892-3. [Online]. Available: https://doi.org/10.1007/s10107-015-0892-3.

[37] NVIDIA, *Cuda c++ programming guide*. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[38] J. Wu, A. Belevich, E. Bendersky, *et al.*, “Gpucc: An open-source gpgpu compiler,” in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO ’16, Barcelona, Spain: Association for Computing Machinery, 2016, pp. 105–116, ISBN: 9781450337786. DOI: 10.1145/2854038.2854041. [Online]. Available: https://doi.org/10.1145/2854038.2854041.

[39] the LLVM community, *Compile cuda with llvm*. [Online]. Available: https://llvm.org/docs/CompileCudaWithLLVM.html.

[40] C.-C. Chang and C.-J. Lin, “Libsvm: A library for support vector machines,” *ACM Trans. Intell. Syst. Technol.*, vol. 2, no. 3, May 2011, ISSN: 2157-6904. DOI: 10.1145/1961189.1961199. [Online]. Available: https://doi.org/10.1145/1961189.1961199.

[41] S. Herrero-Lopez, J. R. Williams, and A. Sanchez, “Parallel multiclass classification using svms on gpus,” in *Proceedings of the 3rd Workshop on general-purpose computation on graphics processing units*, 2010, pp. 2–11.

[42] Q. Li, R. Salman, E. Test, R. Strack, and V. Kecman, “Gpusvm: A comprehensive cuda based support vector machine package,” *Open Computer Science*, vol. 1, no. 4, pp. 387–405, 2011.

[43] H. Graf, E. Cosatto, L. Bottou, I. Dourdanovic, and V. Vapnik, “Parallel support vector machines: The cascade svm,” in *Advances in Neural Information Processing Systems*, L. Saul, Y. Weiss, and L. Bottou, Eds., vol. 17, MIT Press, 2004. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2004/file/d756d3d2b9dac72449a6a6926534558a-Paper.pdf.

[44] Pranav and Contributors, *Argument parser for modern c++ code repository*. [Online]. Available: https://github.com/p-ranav/argparse.

[45] R. A. Fisher, *Iris*, UCI Machine Learning Repository, DOI: https://doi.org/10.24432/C56C76, 1988.

Appendix

Listing 7.1 Generic Vector Implementation

```
#include <assert.h>
#include <functional>

#include <cooperative_groups.h>

#include "cuda_helpers.h"
#include "types.hpp"

namespace cg = cooperative_groups;

using cg::grid_group;
using cg::this_grid;
using cg::this_thread_block;
using cg::thread_block;

namespace types {

template <typename T>
struct cuda_vector;

template <typename T>
struct base_vector {
    idx cols;
    T* data;
    bool view = false;

    __host__ __device__ base_vector() : cols(0), data(nullptr), view(false) {}

    __host__ __device__ base_vector(T* start, T* end) : cols(end - start), data(start), view(true) {}

    __host__ __device__ base_vector(idx _cols) : cols(_cols), data(nullptr), view(false) {}

    __host__ __device__ T& operator[](idx i) {
        if (i >= this->cols) {
            printf("i:%lu, cols %lu\n", i, this->cols);
        }
        assert(i < this->cols);
        return this->data[i];
    }

    __host__ __device__ T& operator[](idx i) const {
        if (i >= this->cols) {
            printf("i:%lu, cols %lu\n", i, this->cols);
        }
        assert(i < this->cols);
        return this->data[i];
    }

    __host__ __device__ T* begin() { return this->data; }
    __host__ __device__ T* end() { return this->data + this->cols - 1; }

    void set(T value) {
        for (idx i = 0; i < this->cols; i++) {
            this->data[i] = value;
        }
    }

    __host__ void mutate(std::function<T(T)> func) {
        for (idx i = 0; i < this->cols; i++) {
            this->data[i] = func(this->data[i]);
        }
    }
    __host__ __device__ void print(const char* msg) const;
};
```

Table 7.1: Raw Data collected for “Training time of serial SMO and parallel GPUSVM”

Training Time	Number of Samples	System	Algorithm
0.164054	1000	WSL	SMO
0.138601	1000	WSL	SMO
0.150266	1000	WSL	SMO
0.158603	1000	WSL	SMO
0.165292	1000	WSL	SMO
0.181241	1000	WSL	SMO
0.168587	1000	WSL	SMO
0.176524	1000	WSL	SMO
0.138604	1000	WSL	SMO
0.165835	1000	WSL	SMO
13.737926	10000	WSL	SMO
13.410943	10000	WSL	SMO
12.427074	10000	WSL	SMO
12.833460	10000	WSL	SMO
12.589262	10000	WSL	SMO
12.775803	10000	WSL	SMO
12.498546	10000	WSL	SMO
14.093719	10000	WSL	SMO
13.116480	10000	WSL	SMO
12.336858	10000	WSL	SMO
0.219552	1000	WSL	GPUSVM
0.173504	1000	WSL	GPUSVM
0.170560	1000	WSL	GPUSVM
0.179104	1000	WSL	GPUSVM
0.170432	1000	WSL	GPUSVM
0.171872	1000	WSL	GPUSVM
0.195488	1000	WSL	GPUSVM
0.176640	1000	WSL	GPUSVM
0.166944	1000	WSL	GPUSVM
0.169760	1000	WSL	GPUSVM
0.191296	10000	WSL	GPUSVM

Table 7.1: Raw Data collected for “Training time of serial SMO and parallel GPUSVM”

Training Time	Number of Samples	System	Algorithm
0.192096	10000	WSL	GPUSVM
0.195968	10000	WSL	GPUSVM
0.263200	10000	WSL	GPUSVM
0.189952	10000	WSL	GPUSVM
0.189664	10000	WSL	GPUSVM
0.194688	10000	WSL	GPUSVM
0.211712	10000	WSL	GPUSVM
0.191200	10000	WSL	GPUSVM
0.191552	10000	WSL	GPUSVM
0.537472	100000	WSL	GPUSVM
0.515712	100000	WSL	GPUSVM
0.513824	100000	WSL	GPUSVM
0.569440	100000	WSL	GPUSVM
0.497536	100000	WSL	GPUSVM
0.502880	100000	WSL	GPUSVM
0.688192	100000	WSL	GPUSVM
0.656480	100000	WSL	GPUSVM
0.562336	100000	WSL	GPUSVM
0.536864	100000	WSL	GPUSVM
3.867552	1000000	WSL	GPUSVM
3.945248	1000000	WSL	GPUSVM
3.844800	1000000	WSL	GPUSVM
4.166016	1000000	WSL	GPUSVM
3.940032	1000000	WSL	GPUSVM
3.842272	1000000	WSL	GPUSVM
3.615136	1000000	WSL	GPUSVM
3.979904	1000000	WSL	GPUSVM
3.818528	1000000	WSL	GPUSVM
3.605600	1000000	WSL	GPUSVM
33.265182	10000000	WSL	GPUSVM
33.515072	10000000	WSL	GPUSVM

Table 7.1: Raw Data collected for “Training time of serial SMO and parallel GPUSVM”

Training Time	Number of Samples	System	Algorithm
33.998623	10000000	WSL	GPUSVM
34.535137	10000000	WSL	GPUSVM
33.702946	10000000	WSL	GPUSVM
33.647678	10000000	WSL	GPUSVM
34.509216	10000000	WSL	GPUSVM
33.876766	10000000	WSL	GPUSVM
32.181438	10000000	WSL	GPUSVM
33.760895	10000000	WSL	GPUSVM
0.083588	1000	Headless	SMO
0.077406	1000	Headless	SMO
0.074697	1000	Headless	SMO
0.083789	1000	Headless	SMO
0.077528	1000	Headless	SMO
0.077948	1000	Headless	SMO
0.077895	1000	Headless	SMO
0.075502	1000	Headless	SMO
0.075551	1000	Headless	SMO
0.075394	1000	Headless	SMO
7.856127	10000	Headless	SMO
7.842031	10000	Headless	SMO
7.796364	10000	Headless	SMO
7.827624	10000	Headless	SMO
7.831180	10000	Headless	SMO
7.828252	10000	Headless	SMO
7.820098	10000	Headless	SMO
7.719477	10000	Headless	SMO
7.830530	10000	Headless	SMO
7.893655	10000	Headless	SMO
796.136292	100000	Headless	SMO
796.660461	100000	Headless	SMO
0.056288	1000	Headless	GPUSVM

Table 7.1: Raw Data collected for “Training time of serial SMO and parallel GPUSVM”

Training Time	Number of Samples	System	Algorithm
0.053792	1000	Headless	GPUSVM
0.059680	1000	Headless	GPUSVM
0.051552	1000	Headless	GPUSVM
0.050080	1000	Headless	GPUSVM
0.059232	1000	Headless	GPUSVM
0.060384	1000	Headless	GPUSVM
0.147168	1000	Headless	GPUSVM
0.144800	1000	Headless	GPUSVM
0.057760	1000	Headless	GPUSVM
0.143808	10000	Headless	GPUSVM
0.062144	10000	Headless	GPUSVM
0.052224	10000	Headless	GPUSVM
0.141504	10000	Headless	GPUSVM
0.438336	10000	Headless	GPUSVM
1.041440	10000	Headless	GPUSVM
0.138304	10000	Headless	GPUSVM
1.194336	10000	Headless	GPUSVM
0.151968	10000	Headless	GPUSVM
0.180928	10000	Headless	GPUSVM
0.268384	100000	Headless	GPUSVM
0.177184	100000	Headless	GPUSVM
0.182304	100000	Headless	GPUSVM
0.161760	100000	Headless	GPUSVM
0.177984	100000	Headless	GPUSVM
0.188192	100000	Headless	GPUSVM
0.176576	100000	Headless	GPUSVM
0.152448	100000	Headless	GPUSVM
0.171328	100000	Headless	GPUSVM
0.218112	100000	Headless	GPUSVM
0.412320	1000000	Headless	GPUSVM
0.397440	1000000	Headless	GPUSVM

Table 7.1: Raw Data collected for “Training time of serial SMO and parallel GPUSVM”

Training Time	Number of Samples	System	Algorithm
0.398624	1000000	Headless	GPUSVM
0.379776	1000000	Headless	GPUSVM
0.379072	1000000	Headless	GPUSVM
0.383168	1000000	Headless	GPUSVM
0.386496	1000000	Headless	GPUSVM
0.507232	1000000	Headless	GPUSVM
0.411840	1000000	Headless	GPUSVM
0.396704	1000000	Headless	GPUSVM
1.907008	10000000	Headless	GPUSVM
2.169120	10000000	Headless	GPUSVM
1.923072	10000000	Headless	GPUSVM
1.879392	10000000	Headless	GPUSVM
1.904576	10000000	Headless	GPUSVM
1.934944	10000000	Headless	GPUSVM
1.852192	10000000	Headless	GPUSVM
1.883136	10000000	Headless	GPUSVM
1.937824	10000000	Headless	GPUSVM
2.091296	10000000	Headless	GPUSVM

Table 7.2: Raw Data collected for “Training time for differing thread count”

Threads	Blocks	Training Time	System
256	10	3.686752	WSL
256	10	3.841952	WSL
256	10	3.623552	WSL
256	10	3.654976	WSL
256	10	3.622208	WSL
256	10	3.642016	WSL
256	10	3.919136	WSL
256	10	3.599264	WSL
256	10	3.640416	WSL

Table 7.2: Raw Data collected for “Training time for differing thread count”

Threads	Blocks	Training Time	System
256	10	3.669472	WSL
128	30	2.922784	WSL
128	30	2.635360	WSL
128	30	2.630272	WSL
128	30	2.605664	WSL
128	30	2.609600	WSL
128	30	2.896512	WSL
128	30	2.691360	WSL
128	30	2.624992	WSL
128	30	2.621408	WSL
128	30	2.604608	WSL
64	60	2.579872	WSL
64	60	2.713184	WSL
64	60	2.898784	WSL
64	60	2.597440	WSL
64	60	2.645600	WSL
64	60	2.640224	WSL
64	60	2.646464	WSL
64	60	2.629792	WSL
64	60	2.647680	WSL
64	60	2.627168	WSL
512	72	0.412800	Headless
512	72	0.367968	Headless
512	72	0.363904	Headless
512	72	0.455648	Headless
512	72	0.391936	Headless
512	72	0.437792	Headless
512	72	0.382048	Headless
512	72	0.385728	Headless
512	72	0.397184	Headless
512	72	0.413632	Headless

Table 7.2: Raw Data collected for “Training time for differing thread count”

Threads	Blocks	Training Time	System
256	144	0.374336	Headless
256	144	0.418240	Headless
256	144	0.401152	Headless
256	144	0.392288	Headless
256	144	0.370464	Headless
256	144	0.369568	Headless
256	144	0.371584	Headless
256	144	0.459456	Headless
256	144	0.375712	Headless
256	144	0.420864	Headless

Table 7.3: Raw Data collected for “Training time for small datasets by algorithm”

Algorithm	Dataset	Training Time	System
SMO	Iris	0.008840	WSL
SMO	Iris	0.009079	WSL
SMO	Iris	0.007134	WSL
SMO	Iris	0.007323	WSL
SMO	Iris	0.010631	WSL
SMO	Iris	0.009631	WSL
SMO	Iris	0.010637	WSL
SMO	Iris	0.008678	WSL
SMO	Iris	0.007629	WSL
SMO	Iris	0.008842	WSL
SMO	Linear 1k	0.119144	WSL
SMO	Linear 1k	0.133577	WSL
SMO	Linear 1k	0.154470	WSL
SMO	Linear 1k	0.128313	WSL
SMO	Linear 1k	0.137159	WSL
SMO	Linear 1k	0.128119	WSL
SMO	Linear 1k	0.115076	WSL

Table 7.3: Raw Data collected for “Training time for small datasets by algorithm”

Algorithm	Dataset	Training Time	System
SMO	Linear 1k	0.145228	WSL
SMO	Linear 1k	0.136425	WSL
SMO	Linear 1k	0.114279	WSL
GPUSVM	Iris	0.643648	WSL
GPUSVM	Iris	0.663008	WSL
GPUSVM	Iris	1.043776	WSL
GPUSVM	Iris	0.747104	WSL
GPUSVM	Iris	0.652000	WSL
GPUSVM	Iris	0.611552	WSL
GPUSVM	Iris	0.626048	WSL
GPUSVM	Iris	0.641760	WSL
GPUSVM	Iris	0.610112	WSL
GPUSVM	Iris	0.630272	WSL
GPUSVM	Linear 1k	0.200192	WSL
GPUSVM	Linear 1k	0.190912	WSL
GPUSVM	Linear 1k	0.170048	WSL
GPUSVM	Linear 1k	0.171424	WSL
GPUSVM	Linear 1k	0.169888	WSL
GPUSVM	Linear 1k	0.171296	WSL
GPUSVM	Linear 1k	0.167328	WSL
GPUSVM	Linear 1k	0.169888	WSL
GPUSVM	Linear 1k	0.183968	WSL
GPUSVM	Linear 1k	0.327616	WSL
SMO	Iris	0.008954	Headless
SMO	Iris	0.009010	Headless
SMO	Iris	0.009594	Headless
SMO	Iris	0.010642	Headless
SMO	Iris	0.005115	Headless
SMO	Iris	0.010198	Headless
SMO	Iris	0.009275	Headless
SMO	Iris	0.009795	Headless

Table 7.3: Raw Data collected for “Training time for small datasets by algorithm”

Algorithm	Dataset	Training Time	System
SMO	Iris	0.006847	Headless
SMO	Iris	0.005070	Headless
SMO	Linear 1k	0.076970	Headless
SMO	Linear 1k	0.075494	Headless
SMO	Linear 1k	0.077067	Headless
SMO	Linear 1k	0.074679	Headless
SMO	Linear 1k	0.079677	Headless
SMO	Linear 1k	0.080738	Headless
SMO	Linear 1k	0.076932	Headless
SMO	Linear 1k	0.076451	Headless
SMO	Linear 1k	0.072776	Headless
SMO	Linear 1k	0.077876	Headless
GPUSVM	Iris	0.218816	Headless
GPUSVM	Iris	0.171680	Headless
GPUSVM	Iris	0.170112	Headless
GPUSVM	Iris	0.171040	Headless
GPUSVM	Iris	0.186432	Headless
GPUSVM	Iris	0.171488	Headless
GPUSVM	Iris	0.168128	Headless
GPUSVM	Iris	0.175040	Headless
GPUSVM	Iris	0.170752	Headless
GPUSVM	Iris	0.172032	Headless
GPUSVM	Linear 1k	0.048672	Headless
GPUSVM	Linear 1k	0.213088	Headless
GPUSVM	Linear 1k	0.047904	Headless
GPUSVM	Linear 1k	0.046816	Headless
GPUSVM	Linear 1k	0.048032	Headless
GPUSVM	Linear 1k	0.047680	Headless
GPUSVM	Linear 1k	0.049472	Headless
GPUSVM	Linear 1k	0.048192	Headless
GPUSVM	Linear 1k	0.048736	Headless

Table 7.3: Raw Data collected for “Training time for small datasets by algorithm”

Algorithm	Dataset	Training Time	System
GPUSVM	Linear 1k	0.047968	Headless

7.1 Full Project

The full project, including the source code, dataset helper scripts and the markdown source of this paper will be available online after publication at this public repository.

Listing 7.2 Generic Vector Implementation cont.

```
template <typename T = math_t>
struct vector : public base_vector<T> {

    vector() = delete;

    // sized constructor
    vector(idx _cols)
        : base_vector<T>(_cols) {
        this->data = new T[this->cols];
    }

    vector(T* start, T* end)
        : base_vector<T>(start, end) {}

    // move constructor
    vector(vector&& other)
        : base_vector<T>(other.cols) { // TODO: check if base_vector() is called automagically
                                       // DONE: it is
        *this = std::move(other);
    }
    // move assignment
    vector& operator=(vector&& other) {
        // puts("vector<T>::move");
        // printf("addr %p\n", this);
        assert(other.view == false);
        delete[] this->data;
        this->data = other.data;
        this->cols = other.cols;
        other.data = nullptr;
        other.cols = 0;
        return *this;
    }

    // copy constructor
    vector(vector& other)
        : base_vector<T>(other.cols) {
        // puts("vector<T>::copy const");
        *this = other;
    }
    // copy assignment
    vector& operator=(vector& other) {
        // puts("vector<T>::copy assign");
        if (this->cols != other.cols || this->data == nullptr) { // don't delete[n] just to new[n]
            delete[] this->data;
            this->data = new T[other.cols];
            // printf("%p\n", &this->data);
            this->cols = other.cols;
        }
        memcpy(this->data, other.data, sizeof(T) * other.cols);
        return *this;
    }

    // copy conversion constructor
    vector(cuda_vector<T>& other)
        : base_vector<T>(other.cols) {
        *this = other;
    }

    // copy convert
    vector<T>& operator=(cuda_vector<T>& other) {
        if (this->cols != other.cols || this->data == nullptr) {
            delete[] this->data;
            this->data = new T[other.cols];
            this->cols = other.cols;
        }
        cudaErr(cudaMemcpy(this->data, other.data, sizeof(T) * other.cols, cudaMemcpyDeviceToHost));
        return *this;
    }

    ~vector() {
        if (!this->view) {
            delete[] this->data;
        }
    }
};
```

Listing 7.3 Generic Vector Implementation cont.

```
template <typename T>
struct cuda_vector : public base_vector<T> {
    // sized constructor
    cuda_vector(idx _cols)
        : base_vector<T>(_cols) {
        cudaErr(cudaMalloc(&this->data, sizeof(T) * this->cols));
    }
    __host__ __device__ cuda_vector(T* start, T* end)
        : base_vector<T>(start, end) {}
    // move constructor
    cuda_vector(cuda_vector&& other) {
        *this = std::move(other);
    }
    // move assignment
    cuda_vector& operator=(cuda_vector&& other) {
        assert(other.view == false);
        cudaErr(cudaFree(this->data));
        this->data = other.data;
        this->cols = other.cols;
        other.data = nullptr;
        other.cols = 0;
        return *this;
    }

    // copy constructor
    cuda_vector(cuda_vector& other)
        : base_vector<T>(other.cols) {
        *this = other;
    }
    // copy assignment
    cuda_vector& operator=(cuda_vector& other) {
        // puts("vector<T>::copy");
        if (this->cols != other.cols || this->data == nullptr) { // don't delete[n] just to new[n]
            cudaErr(cudaFree(this->data));
            cudaErr(cudaMalloc(&this->data, sizeof(T) * other.cols));
            // printf("%p\n", &this->data);
            this->cols = other.cols;
        }
        cudaErr(cudaMemcpy(this->data, other.data, sizeof(T) * other.cols, cudaMemcpyDeviceToDevice));
        return *this;
    }

    // copy conversion constructor
    cuda_vector(vector<T>& other)
        : base_vector<T>(other.cols) {
        *this = other;
    }

    // conversion
    cuda_vector& operator=(vector<T>& other) {
        if (this->cols != other.cols || this->data == nullptr) {
            cudaErr(cudaFree(this->data));
            cudaErr(cudaMalloc(&this->data, sizeof(T) * other.cols));
            this->cols = other.cols;
        }
        cudaErr(cudaMemcpy(this->data, other.data, sizeof(T) * other.cols, cudaMemcpyHostToDevice));
        return *this;
    }

    __host__ __device__ ~cuda_vector() {
#ifndef __CUDA_ARCH__
        if (!this->view) {
            cudaErr(cudaFree(this->data));
        }
#endif
    }
```

Listing 7.4 Generic Vector Implementation cont.

```
    template <class F>
    __device__ void mutate(F func) {
        unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
        unsigned int stride = blockDim.x * gridDim.x;
        grid_group grid = this_grid();

        for (idx i = tid; i < this->cols; i += stride) {
            this->data[i] = func(i);
        }
        grid.sync();
    }

    __device__ void set(T value) {
        unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
        unsigned int stride = blockDim.x * gridDim.x;
        grid_group grid = this_grid();

        // if (tid == 0)
        //     printf("[%d]: [%p] = value \n", tid, this);
        // printf("[%d]: [%i] = value \n", tid, this->cols);
        for (idx i = tid; i < this->cols; i += stride) {
            // if (tid == 0)
            //     printf("[%d]: set [%i] = value \n", tid, i);
            this->data[i] = value;
        }
        grid.sync();
    }
};

typedef math_t (*Kernel)(base_vector<math_t>, base_vector<math_t>);

// using Kernel = std::function<number(vector<number>, vector<number>)>;

template <>
inline void base_vector<int>::print(const char* msg) const {
    for (idx i = 0; i < this->cols; i++) {
        printf("%s[%zu]: %*d\n", msg, i, PRINT_DIGITS, this->data[i]);
    }
}

template <>
inline void base_vector<double>::print(const char* msg) const {
    for (idx i = 0; i < this->cols; i++) {
        printf("%s[%zu]: %*.*f\n", msg, i, PRINT_DIGITS, PRINT_AFTER, this->data[i]);
    }
}

template <typename T>
void _printd(base_vector<T>& vec, const char* msg) {
    vec.print(msg);
}

} // namespace types
#endif
```

Listing 7.5 Generic Matrix Implementation

```
#ifndef MATRIX_HPP
#define MATRIX_HPP 1

#include "types.hpp"
#include "vector.hpp"

namespace types {

template <typename T>
struct cuda_matrix;

template <typename T>
struct base_matrix {
    idx rows;
    idx cols;
    T* data;
    base_matrix()
        : rows(0),
          cols(0),
          data(nullptr) {}
    base_matrix(idx _rows, idx _cols)
        : rows(_rows),
          cols(_cols),
          data(nullptr) {}

    T* begin() {
        return this->data;
    }

    T* end() {
        return this->data + rows * cols;
    }

    auto shape() {
        return std::make_tuple(rows, cols);
    }
};
```

Listing 7.6 Generic Matrix Implementation cont.

```
template <typename T>
struct matrix : public base_matrix<T> {

    // sized constructor
    matrix(idx _rows, idx _cols)
        : base_matrix<T>(_rows, _cols) {
        this->data = new T[_cols * _rows];
    }
    // move constructor
    matrix(matrix&& other)
        : base_matrix<T>(other.rows, other.cols) {
        *this = std::move(other);
    }

    // move assignment
    matrix& operator=(matrix&& other) {
        delete[] this->data;
        this->data = other.data;
        this->cols = other.cols;
        this->rows = other.rows;
        other.data = nullptr;
        return *this;
    }

    matrix(matrix& other)
        : base_matrix<T>(other.rows, other.cols) {
        *this = other;
    }

    matrix& operator=(matrix& other) {
        if (this->cols * this->rows != other.cols * other.rows) {
            delete[] this->data;
            this->data = new T[other.cols * other.rows];
            this->cols = other.cols;
            this->rows = other.cols;
        }
        memcpy(this->data, other.data, other.rows * other.cols * sizeof(T));
        return *this;
    }

    matrix(cuda_matrix<T>& other)
        : base_matrix<T>(other.rows, other.cols) {
        *this = other;
    }

    matrix& operator=(cuda_matrix<T>& other) {
        if (this->cols * this->rows != other.cols * other.rows || this->data == nullptr) {
            free(this->data);
            this->data = new T[other.cols * other.rows];
            this->cols = other.cols;
            this->rows = other.rows;
        }
        cudaErr(cudaMemcpy(this->data, other.data, sizeof(T) * other.cols * other.rows, cudaMemcpyDeviceToHost));
        return *this;
    }

    ~matrix() {
        // printf("~matrix: %p\n", this);
        delete[] this->data;
    }

    // returns a vector which does not deallocate it's data, since it's owned by this matrix
    vector<T> operator[](idx index) {
        assert(index < this->rows);
        return vector<T>(&(this->data[index * this->cols]), &(this->data[index * this->cols + this->cols]));
    }
    vector<T> operator[](idx index) const {
        assert(index < this->rows);
        return vector<T>(&(this->data[index * this->cols]), &(this->data[index * this->cols + this->cols]));
    }

    void print() {
        for (idx i = 0; i < this->rows; i++) {
            for (idx j = 0; j < this->cols; j++) {
                printf("%*.*f ", PRINT_DIGITS, PRINT_AFTER, this->data[i * this->cols + j]);
            }
            puts("");
        }
    }
};
```

Listing 7.7 Generic Matrix Implementationt cont.

```
template <typename T>
void inline _printd(matrix<T>& mat, const char* msg) {
    puts(msg);
    mat.print();
}

template <typename T>
struct cuda_matrix : base_matrix<T> {

    // sized constructor
    cuda_matrix(idx _rows, idx _cols)
        : base_matrix<T>(_rows, _cols) {
        cudaErr(cudaMalloc(&this->data, sizeof(T) * _cols * _rows));
    }
    // move constructor
    cuda_matrix(matrix<T>&& other)
        : base_matrix<T>(other.rows, other.cols) {
        *this = std::move(other);
    }

    // move assignment
    cuda_matrix& operator=(matrix<T>&& other) {
        cudaErr(cudaFree(this->data));
        this->data = other.data;
        this->cols = other.cols;
        this->rows = other.rows;
        other.data = nullptr;
        return *this;
    }

    cuda_matrix(matrix<T>& other)
        : base_matrix<T>(other.rows, other.cols) {
        *this = other;
    }

    cuda_matrix& operator=(matrix<T>& other) {
        if (this->cols * this->rows != other.cols * other.rows || this->data == nullptr) {
            cudaErr(cudaFree(this->data));
            cudaErr(cudaMalloc(&this->data, sizeof(T) * other.cols * other.rows));
            this->cols = other.cols;
            this->rows = other.rows;
        }
        cudaErr(cudaMemcpy(this->data, other.data, sizeof(T) * other.cols * other.rows, cudaMemcpyHostToDevice));
        return *this;
    }

    ~cuda_matrix() {
        // printf("~matrix: %p\n", this);
        cudaErr(cudaFree(this->data));
    }

    // returns a vector which does not deallocate it's data, since it's owned by this cuda_matrix
    __device__ cuda_vector<T> operator[](idx index) {
        return cuda_vector<T>(&(this->data[index * this->cols]), &(this->data[index * this->cols + this->cols]));
    }
    __device__ cuda_vector<T> operator[](idx index) const {
        return cuda_vector<T>(&(this->data[index * this->cols]), &(this->data[index * this->cols + this->cols]));
    }
};

} // namespace types
#endif
```

Listing 7.8 SMO Implementation Outer Loop

```
while (numChanged > 0 || examineAll) {
    if (epochs % 100 == 0) {
        printf(".\n");
        std::flush(std::cout);
    }
    if (false && epochs && epochs % 1000 == 0) {
        math_t avg_error = 0;
        for (auto e : error) {
            avg_error += e;
        }
        avg_error = avg_error / static_cast<math_t>(error.cols);

        printf("\nContinue training? [Y/n]\n");
        printf("Already trained for %d epochs.\n", epochs);
        printf("Average error on training set: %f\n", avg_error);
        int c = getchar();
        if (c == 'n') {
            puts("Quit training!");
            break;
        }
        if (c != '\n') {
            getchar();
        }
    }
    numChanged = 0;

    if (examineAll) {
        // puts("examine all");
        // loop i_1 over all training examples
        for (idx i2 = 0; i2 < x.rows; i2++) {
            numChanged += examineExample(i2);
        }
    } else {
        // puts("examine some");
        // loop i_1 over examples for which alpha is nor 0 nor Cost
        for (idx i2 = 0; i2 < x.rows; i2++) {
            if (a[i2] != 0.0 || a[i2] != C) {
                numChanged += examineExample(i2);
            }
        }
    }
    if (examineAll) {
        examineAll = false;
    } else if (numChanged == 0) {
        puts("None changed, so examine all!");
        examineAll = true;
    }
    epochs++;
    if (epochs >= 10000) {
        puts("Max iteration limit reached!");
        break;
    }
}
printf("Done!\nTrained for %d epochs.\n", epochs);
auto end = std::chrono::steady_clock::now();
float elapsed_seconds = std::chrono::duration_cast<std::chrono::duration<float>>(end - start).count();
return elapsed_seconds;
```

Listing 7.9 SMO Implementation Inner Loop

```
int examineExample(idx i2) {
    // printf("examine example %zu\n", i2);

    // lookup error E2 for i_2 in error cache
    math_t E2 = error[i2];

    math_t r2 = E2 * y[i2];

    // if the error is within tolerance and the a is outside of (0, C)
    // don't change anything for this i_2
    if ((r2 < -tol && a[i2] < C) || (r2 > tol && a[i2] > 0)) {
        // number of non-zero & non-C alphas
        int non_zero_non_c = 0;
        for (idx i = 0; i < a.cols; i++) {
            if (a[i] < types::epsilon || fabs(a[i] - C) < types::epsilon) {
                continue;
            }
            non_zero_non_c++;
            if (non_zero_non_c > 1) { // no need to count them all
                break;
            }
        }
        if (non_zero_non_c > 1) {
            idx i1 = second_choice_heuristic(E2);
            if (takeStep(i1, i2) == 1) {
                return 1;
            }
        }

        // in the following 2 scopes
        // iters makes sure we go over all i_1
        // i_1 is the current i_1, starting from a random one, increasing until starting_i_1 - 1
        // i_1 wraps around if > a.cols

        // loop i_1 over all non-zero non-C a, starting at random point
        {

            idx iters = 0;
            idx i1 = 0;
            do {
                i1 = static_cast<idx>(rand()) % a.cols;
            } while (i1 == i2);

            do {
                if (fabs(a[i1]) < types::epsilon || fabs(a[i1] - C) < types::epsilon) {
                    continue;
                }
                if (takeStep(i1, i2) == 1) {
                    return 1;
                }
            } while (i1 = (i1 + 1) % a.cols, iters++, iters < a.cols);
        }

        {
            idx iters = 0;
            idx i1 = 0;
            do {
                i1 = static_cast<idx>(rand()) % a.cols;
            } while (i1 == i2);

            do {
                if (takeStep(i1, i2) == 1) {
                    return 1;
                }
            } while (i1 = (i1 + 1) % a.cols, iters++, iters < a.cols);
        }
    }

    return 0;
}
```

Listing 7.10 SMO Implementation Step

```
int takeStep(idx i1, idx i2) {
    // getchar();
    // printf("    takeStep %zu %zu\n", i1, i2);
    math_t sign = y[i1] * y[i2];
    math_t L = 0, H = 0;

    // find low and high
    if (y[i1] != y[i2]) {
        L = max(0, a[i2] - a[i1]);
        H = min(C, C + a[i2] - a[i1]);
    } else {
        L = max(0, a[i1] + a[i2] - C);
        H = min(C, a[i1] + a[i2]);
    }
    if (fabs(H - L) < types::epsilon) {
        // puts("      Low equals High");
        return 0;
    }

    // second derivative (f'')
    math_t eta = 2 * Kernel(x[i1], x[i2]) - Kernel(x[i1], x[i1]) - Kernel(x[i2], x[i2]);

    math_t a_1 = 0, a_2 = 0;
    if (eta < 0) { // if ("under usual circumstances") eta is negative
        // puts("      by error");
        // error on training examples i_1 and i_2
        math_t E1 = error[i1];
        math_t E2 = error[i2];

        // new a_2
        a_2 = a[i2] - (y[i2] * (E1 - E2)) / eta + types::epsilon;

        // clip a_2
        if (a_2 > H) {
            a_2 = H;
        } else if (a_2 < L) {
            a_2 = L;
        }
    } else {
        // TODO: eq 12.21 again for = f^{old}(x_i) ...
        // puts("      by objective eval");
        // puts("      skipping..");
        return 0;
        auto WL = eval_objective_func_at(i1, i2, L);
        auto WH = eval_objective_func_at(i1, i2, H);

        if (WL > WH) {
            a_2 = WL;
        } else {
            a_2 = WH;
        }
    }
    a_1 = a[i1] + sign * (a[i2] - a_2);

    // if the difference is small, don't bother
    if (fabs(a[i1] - a_1) < diff_tol) {
        // puts("small diff");
        return 0;
    }

    // puts("      changed\n");

    a[i1] = a_1;
    a[i2] = a_2;
    compute_w();
    b = compute_b();

    error[i1] = predict_on(i1) - y[i1];
    error[i2] = predict_on(i2) - y[i2];

    return 1;
}
```

Listing 7.11 GPUSVM Grid Reduction

```
__device__ idx_tuple argMin(size_t shared_halfpoint) {
    unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int stride = blockDim.x * gridDim.x;
    thread_block threads = this_thread_block();
    grid_group blocks = this_grid();

    extern __shared__ char shared_memory[];

    idx* sindx = reinterpret_cast<idx*>(shared_memory);
    math_t* sdata = reinterpret_cast<math_t*>(shared_memory + (sizeof(idx) * shared_halfpoint));

    __shared__ math_t block_max;
    __shared__ idx block_max_i;
    __shared__ math_t block_min;
    __shared__ idx block_min_i;
    // init block locals
    if (threadIdx.x == 0) {
        block_max = -types::MATH_T_MAX;
        block_max_i = 0;
        block_min = +types::MATH_T_MAX;
        block_min_i = 0;
    }

    math_t cur_max;
    math_t cur_min;
    idx min_i = tid;
    idx max_i = tid;
    if (tid < a.cols) {
        cur_min = a[tid];
        // thread local arg min|max
        for (idx i = tid + stride; i < a.cols; i += stride) {
            // TODO: Take into account indices up, low and both
            auto kind = indices[i];
            auto tmp = a[i]; // save to local, so it's accessed only once
            if (kind == UP || kind == BOTH) {
                if (tmp < cur_min) {
                    cur_min = tmp;
                    min_i = i;
                }
            }
            if (kind == LOW || kind == BOTH) {
                if (tmp > cur_min) {
                    cur_max = tmp;
                    max_i = i;
                }
            }
        }
    } else {
        cur_max = -types::MATH_T_MAX;
        cur_min = +types::MATH_T_MAX;
    }

    // load into to block shared memory
    sdata[threadIdx.x] = cur_min;
    sindx[threadIdx.x] = min_i;
    sdata[threadIdx.x + blockDim.x] = cur_max;
    sindx[threadIdx.x + blockDim.x] = max_i;
    threads.sync();

    // block reduce into sdata[0] and sindx[0]
    for (idx offset = 1; offset < blockDim.x; offset *= 2) {
        idx index = 2 * offset * threadIdx.x;

        if (index < blockDim.x) {
            // min
            if (sdata[index + offset] < sdata[index]) {
                sdata[index] = sdata[index + offset];
                sindx[index] = sindx[index + offset];
            }
            // max (stored offset by blockDim.x)
            if (sdata[blockDim.x + index + offset] > sdata[blockDim.x + index]) {
                sdata[blockDim.x + index] = sdata[blockDim.x + index + offset];
                sindx[blockDim.x + index] = sindx[blockDim.x + index + offset];
            }
        }
    }
```

Listing 7.12 GPUSVM Grid Reduction cont.

```
    threads.sync();
    // write block result to device global
    if (threadIdx.x == 0) {
        ddata[blockIdx.x] = sdata[0];
        dindx[blockIdx.x] = sindx[0];
        ddata[blockDim.x + blockIdx.x] = sdata[blockDim.x + 0];
        dindx[blockDim.x + blockIdx.x] = sindx[blockDim.x + 0];
    }

    blocks.sync();

    // perform reduction of block results,
    // like above but for block results :^)
    if (blockIdx.x == 0) {
        // copy device globals to block shared memory
        if (threadIdx.x < gridDim.x) {
            sdata[threadIdx.x] = ddata[threadIdx.x];
            sindx[threadIdx.x] = dindx[threadIdx.x];
            sdata[blockDim.x + threadIdx.x] = ddata[blockDim.x + threadIdx.x];
            sindx[blockDim.x + threadIdx.x] = dindx[blockDim.x + threadIdx.x];
        }
        threads.sync();
        for (idx offset = 1; offset < blockDim.x; offset *= 2) {
            idx index = 2 * offset * threadIdx.x;

            if (index < blockDim.x) {
                if (sdata[index + offset] < sdata[index]) {
                    sdata[index] = sdata[index + offset];
                    sindx[index] = sindx[index + offset];
                }
                if (sdata[blockDim.x + index + offset] > sdata[blockDim.x + index]) {
                    sdata[blockDim.x + index] = sdata[blockDim.x + index + offset];
                    sindx[blockDim.x + index] = sindx[blockDim.x + index + offset];
                }
            }
        }
        threads.sync();
    }

    // write to global memory
    if (blockIdx.x + threadIdx.x == 0) { // will the real tid 0 plz stand up
        dindx[0] = sindx[0];
        dindx[1] = sindx[blockDim.x + 0];
    }

    blocks.sync();

    return {.a = dindx[0], .b = dindx[1]};
}
```

Listing 7.13 GPUSVM training device-side code

```
__device__ void train_device(size_t shared_memory) {
    a.set(0);

    unsigned int tid = blockDim.x * blockIdx.x + threadIdx.x;
    unsigned int stride = blockDim.x * gridDim.x;
    grid_group blocks = this_grid();
    thread_block threads = this_thread_block();
    // IDEA: block locals ?

    // thread locals
    math_t K_lo_lo;
    math_t K_up_up;
    math_t K_lo_up;
    math_t eta;
    math_t a_lo_new;
    math_t a_up_new;
    math_t a_lo;
    math_t a_up;
    idx lo;
    idx up;

    // initialize indices
    for (idx i = tid; i < indices.cols; i += stride) { // :^)
        if (0 < a[i] && a[i] < C) {
            indices[i] = BOTH;
            continue;
        }
        if ((y[i] == 1 && a[i] == 0) || (y[i] == -1 && a[i] == C)) {
            indices[i] = UP;
        } else {
            indices[i] = LOW;
        }
        // printf("[%d]: indices[%lu] = %s\n", tid, i, indices[i] == UP ? "UP" : indices[i] == LO ? "LO" : "BOTH");
    }

    // initialize error
    // this->error.mutate([this] __device__(idx _i) -> idx { return -this->y[_i]; });
    for (idx i = tid; i < error.cols; i += stride) {
        error[i] = -y[i];
    }
    blocks.sync();

    // pick b_up and _lo for the first time:
    bool picked_lo = false;
    bool picked_up = false;
    if (tid == 0) {
        for (idx i = 0; i < y.cols; i += 1) {
            if (y[i] < 0) {
                if (!picked_lo) {
                    lo = i;
                    picked_lo = true;
                    if (picked_up) {
                        break;
                    }
                }
            } else {
                if (!picked_up) {
                    up = i;
                    picked_up = true;
                    if (picked_lo) {
                        break;
                    }
                }
            }
        }
```

Listing 7.14 GPUSVM training device-side code cont.

```
    K_lo_lo = Kernel(x[lo], x[lo]);
    K_up_up = Kernel(x[up], x[up]);
    K_lo_up = Kernel(x[lo], x[up]);
    eta = K_lo_lo + K_up_up - 2 * K_lo_up;
    math_t a_new = 2 / eta;
    // write to global memory
    dev_a_up_new = a_new;
    dev_a_lo_new = a_new;
    a[lo] = a_new;
    a[up] = a_new;
    dev_lo = lo;
    dev_up = up;

    // recalculate indices for lo and up
    indices[up] = compute_type_for_index(up);
    indices[lo] = compute_type_for_index(lo);
}
blocks.sync();
// read from global
a_lo_new = dev_a_lo_new;
a_up_new = dev_a_up_new;
lo = dev_lo;
up = dev_up;

for (idx i = tid; i < error.cols; i += stride) {
    error[i] = error[i] - a_lo_new * Kernel(x[lo], x[i]) + a_up_new * Kernel(x[up], x[i]);
}

blocks.sync();
while (dev_b_lo > dev_b_up + 2 * tol) {

    if (tid == 0) {
        math_t sign = y[up] * y[lo];

        K_lo_lo = Kernel(x[lo], x[lo]);
        K_up_up = Kernel(x[up], x[up]);
        K_lo_up = Kernel(x[lo], x[up]);

        eta = K_lo_lo + K_up_up - 2 * K_lo_up;

        // update a_I_up , a_I_lo

        a_up = a[up];
        a_lo = a[lo];

        a_up_new = a_up + (y[up] * (error[lo] - error[up])) / eta + types::epsilon;

        // clip new a_up
        if (a_up_new > C) {
            a_up_new = C;
        } else if (a_up_new < 0) {
            a_up_new = 0;
        }

        a_lo_new = a_lo + sign * (a_up - a_up_new);
        // write to global
        dev_a_up_new = a_up_new;
        dev_a_lo_new = a_lo_new;
    }
    blocks.sync();
    a_lo_new = dev_a_lo_new;
    a_up_new = dev_a_up_new;
    a_lo = dev_a_lo;
    a_up = dev_a_up;
```

Listing 7.15 GPUSVM training device-side code cont.

```
        // recalculate error
        for (idx i = tid; i < error.cols; i += stride) {
            error[i] = error[i] + (a_lo_new - a_lo) * y[lo] * Kernel(x[lo], x[i]) +
                       (a_up_new - a_up) * y[up] * Kernel(x[up], x[i]);
        }
        blocks.sync();

        if (tid == 0) {
            // set new alphas
            a[lo] = a_lo_new;
            a[up] = a_up_new;
            // recompute index type for up and low
            indices[lo] = compute_type_for_index(lo);
            indices[up] = compute_type_for_index(up);
        }
        blocks.sync();

        auto result = argMin(shared_memory);
        up = result.a;
        lo = result.b;

        if (tid == 0) {
            dev_b_lo = error[lo];
            dev_b_up = error[up];
        }
        blocks.sync();
    }
    b = (dev_b_lo + dev_b_up) / 2;
}
```

Listing 7.16 Linearly separable dataset generation script

```
#!/usr/bin/env python3
# import libraries
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
import sys

sizes = {
        "1k": 1000,
        "10k": 10_000,
        "100k": 100_000,
        "1M": 1_000_000,
}

for name, size in sizes.items():
    filename= "linear" + name + ".data"
    print(filename)
    with open(filename, "w") as sys.stdout:
        # generate a 2-class classification problem with 1,000 data points,
        # where each data point is a 2-D feature vector
        (X, Y) = make_blobs(n_samples=size, n_features=3, centers=2,
                        cluster_std=1.5, random_state=1)

        for x, y in zip(X, Y):
            for v in x:
                print(str(v) + ";", end='')
            print(y)
```